ezdxf Documentation

Release 1.0.3

Manfred Moitzi

Jul 21, 2023

CONTENTS

Welcome! This is the documentation for ezdxf release 1.0.3, last updated Jul 21, 2023.

- *ezdxf* is a Python package to create new DXF documents and read/modify/write existing DXF documents
- MIT-License
- the intended audience are programmers
- requires at least Python 3.7
- OS independent
- tested with CPython and pypy3
- has type annotations and passes `mypy --ignore-missing-imports -p ezdxf` successful
- additional required packages for the core package without add-ons: typing_extensions, pyparsing
- read/write/new support for DXF versions: R12, R2000, R2004, R2007, R2010, R2013 and R2018
- additional read-only support for DXF versions R13/R14 (upgraded to R2000)
- additional read-only support for older DXF versions than R12 (upgraded to R12)
- read/write support for ASCII DXF and Binary DXF
- retains third-party DXF content
- optional C-extensions for CPython are included in the binary wheels, available on PyPI for Windows, Linux and macOS

CHAPTER

ONE

INCLUDED EXTENSIONS

Additional packages required for these add-ons are not automatically installed during the *basic* setup, for more information about the setup & dependencies visit the documentation.

- `drawing` add-on to visualise and convert DXF files to images which can be saved as PNG, PDF or SVG files
- `r12writer` add-on to write basic DXF entities direct and fast into a DXF R12 file or stream
- `iterdxf` add-on to iterate over DXF entities from the modelspace of huge DXF files (> 5GB) which do not fit into memory
- `importer` add-on to import entities, blocks and table entries from another DXF document
- `dxf2code` add-on to generate Python code for DXF structures loaded from DXF documents as starting point for parametric DXF entity creation
- `acadctb` add-on to read/write *Plot Style Files (CTB/STB)*
- `pycsg` add-on for Constructive Solid Geometry (CSG) modeling technique
- `MTextExplode` add-on for exploding MTEXT entities into single-line TEXT entities
- `text2path` add-on to convert text into outline paths
- `geo` add-on to support the __geo_interface__
- `meshex` add-on for exchanging meshes with other tools as STL, OFF or OBJ files
- `openscad` add-on, an interface to OpenSCAD
- `odafc` add-on, an interface to the ODA File Converter to read and write DWG files

CHAPTER

TWO

WEBSITE

https://ezdxf.mozman.at/

CHAPTER

THREE

DOCUMENTATION

Documentation of development version at https://ezdxf.mozman.at/docs

Documentation of latest release at http://ezdxf.readthedocs.io/

CHAPTER

FOUR

SOURCE CODE & FEEDBACK

Source Code: http://github.com/mozman/ezdxf.git

Issue Tracker: http://github.com/mozman/ezdxf/issues

Forum: https://github.com/mozman/ezdxf/discussions

CHAPTER

FIVE

QUESTIONS AND ANSWERS

Please post questions at the forum or stack overflow to make answers available to other users as well.

CHAPTER

SIX

CONTENTS

6.1 Introduction

6.1.1 What is ezdxf

Ezdxf is a Python interface to the *DXF* (drawing interchange file) format developed by Autodesk, *ezdxf* allows developers to read and modify existing DXF documents or create new DXF documents.

The main objective in the development of *ezdxf* was to hide complex DXF details from the programmer but still support most capabilities of the *DXF* format. Nevertheless, a basic understanding of the DXF format is required, also to understand which tasks and goals are possible to accomplish by using the DXF format.

Not all DXF features are supported yet, but additional features will be added in the future gradually.

Ezdxf is also a replacement for the outdated dxfwrite and dxfgrabber packages but with different APIs, for more information see also: *What is the Relationship between ezdxf, dxfwrite and dxfgrabber?*

6.1.2 What ezdxf can't do

- *ezdxf* is not a DXF converter: *ezdxf* can not convert between different DXF versions, if you are looking for an appropriate application, try the free ODAFileConverter from the Open Design Alliance, which converts between different DXF version and also between the DXF and the DWG file format.
- *ezdxf* is not a CAD file format converter: *ezdxf* can not convert DXF files to other CAD formats such as DWG
- *ezdxf* is not a CAD kernel and does not provide high level functionality for construction work, it is just an interface to the DXF file format. If you are looking for a CAD kernel with Python scripting support, look at FreeCAD.

6.1.3 Supported Python Versions

Ezdxf requires at least Python 3.7 and will be tested with the latest stable CPython version and the latest stable release of pypy3 during development.

Ezdxf is written in pure Python with optional Cython implementations of some low level math classes and requires only *pyparser* and *typing_extensions* as additional library beside the Python Standard Library. *Pytest* is required to run the unit and integration tests. Data to run the stress and audit test can not be provided, because I don't have the rights for publishing these DXF files.

6.1.4 Supported Operating Systems

Ezdxf is OS independent and runs on all platforms which provide an appropriate Python interpreter (>=3.7).

6.1.5 Supported DXF Versions

Version	AutoCAD Release
AC1009	AutoCAD R12
AC1012	AutoCAD R13 -> R2000
AC1014	AutoCAD R14 -> R2000
AC1015	AutoCAD R2000
AC1018	AutoCAD R2004
AC1021	AutoCAD R2007
AC1024	AutoCAD R2010
AC1027	AutoCAD R2013
AC1032	AutoCAD R2018

Ezdxf also reads older DXF versions but saves it as DXF R12.

6.1.6 Embedded DXF Information of 3rd Party Applications

The DXF format allows third-party applications to embed application-specific information. *Ezdxf* manages DXF data in a structure-preserving form, but for the price of large memory requirement. Because of this, processing of DXF information of third-party applications is possible and will retained on rewriting.

6.1.7 License

Ezdxf is licensed under the very liberal MIT-License.

6.2 Setup & Dependencies

The primary goal is to keep the dependencies of the *core* package as small as possible. The add-ons are not part of the core package and can therefore use as many packages as needed. The only requirement for these packages is an easy way to install them on *Windows*, *Linux* and *macOS*, preferably as:

```
pip3 install ezdxf
```

The pyparsing package and the typing_extensions are the only hard dependency and will be installed automatically by *pip3*!

The minimal required Python version is determined by the latest stable version of pypy3 and the Python version deployed by the Raspberry Pi OS, which would be Python 3.9 in 2022, but Python 3.7 will be kept as the minimal version for the 1.0 release.

6.2.1 Basic Installation

The most common case is the installation by *pip3* including the optional C-extensions from PyPI as binary wheels:

```
pip3 install ezdxf
```

6.2.2 Installation with Extras

To use all features of the drawing add-on, add the [draw] tag:

```
pip3 install ezdxf[draw]
```

Tag	Additional Installed Packages
[draw]	Matplotlib, PySide6, Pillow
[draw5]	Matplotlib, PyQt5, Pillow (use only if PySide6 is not available)
[test]	geomdl, pytest
[dev]	setuptools, wheel, Cython + [test]
[all]	[draw] + [test] + [dev]
[all5]	[draw5] + [test] + [dev] (use only if PySide6 is not available)

6.2.3 Binary Wheels

Ezdxf includes some C-extensions, which will be deployed automatically at each release to PyPI as binary wheels to *PyPI*:

- *Windows*: only amd64 packages
- *Linux*: manylinux and musllinux packages for x86_64 & aarch64
- *macOS*: x86_64, arm64 and universal packages

The wheels are created by the continuous integration (CI) service provided by GitHub and the build container cibuildwheel provided by PyPA the Python Packaging Authority. The workflows are kept short and simple, so my future me will understand what's going on and they are maybe also helpful for other developers which do not touch CI services every day.

The C-extensions are disabled for pypy3, because the JIT compiled code of pypy is much faster than the compiled C-extensions.

6.2.4 Disable C-Extensions

It is possible to disable the C-Extensions by setting the environment variable `EZDXF_DISABLE_C_EXT` to `1` or `true`:

```
set EZDXF_DISABLE_C_EXT=1
```

or on Linux:

```
export EZDXF_DISABLE_C_EXT=1
```

This is has to be done **before** anything from *ezdxf* is imported! If you are working in an interactive environment, you have to restart the interpreter.

6.2.5 Installation from GitHub

Install the latest development version by *pip3* from GitHub:

```
pip3 install git+https://github.com/mozman/ezdxf.git@master
```

6.2.6 Build and Install from Source

This is only required if you want the compiled C-extensions, the *ezdxf* installation by *pip* from the source code package works without the C-extension but is slower. There are binary wheels available on PyPi which included the compiled C-extensions.

Windows 10

Make a build directory and a virtual environment:

```
mkdir build
cd build
py -m venv py310
py310/Scripts/activate.bat
```

A working C++ compiler setup is required to compile the C-extensions from source code. Windows users need the build tools from Microsoft: https://visualstudio.microsoft.com/de/downloads/

Download and install the required Visual Studio Installer of the community edition and choose the option: *Visual Studio Build Tools 20.*.

Install required packages to build and install ezdxf with C-extensions:

```
pip3 install setuptools wheel cython
```

Clone the GitHub repository:

```
git clone https://github.com/mozman/ezdxf.git
```

Build and install ezdxf from source code:

```
cd ezdxf
pip3 install .
```

Check if the installation was successful:

```
python3 -m ezdxf -V
```

The *ezdxf* command should run without a preceding *python3 -m*, but calling the launcher through the interpreter guarantees to call the version which was installed in the venv if there exist a global installation of *ezdxf* like in my development environment.

The output should look like this:

```
ezdxf 0.17.2b4 from D:\Source\build\py310\lib\site-packages\ezdxf
Python version: 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [MSC v.1929 64␣
↪bit (AMD64)]
using C-extensions: yes
using Matplotlib: no
```

To install optional packages go to section: *Install Optional Packages*

To run the included tests go to section: *Run the Tests*

WSL & Ubuntu

I use sometimes the Windows Subsystem for Linux (WSL) with Ubuntu 20.04 LTS for some tests (how to install WSL).

By doing as fresh install on *WSL & Ubuntu*, I encountered an additional requirement, the *build-essential* package adds the required C++ support:

```
sudo apt install build-essential
```

The system Python 3 interpreter has the version 3.8 (in 2021), but I will show in a later section how to install an additional newer Python version from the source code:

```
cd ~
mkdir build
cd build
python3 -m venv py38
source py38/bin/activate
```

Install *Cython* and *wheel* in the venv to get the C-extensions compiled:

```
pip3 install cython wheel
```

Clone the GitHub repository:

```
git clone https://github.com/mozman/ezdxf.git
```

Build and install ezdxf from source code:

```
cd ezdxf
pip3 install .
```

Check if the installation was successful:

```
python3 -m ezdxf -V
```

The output should look like this:

```
ezdxf 0.17.2b4 from /home/mozman/src/py38/lib/python3.8/site-packages/ezdxf
Python version: 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0]
using C-extensions: yes
using Matplotlib: no
```

To install optional packages go to section: *Install Optional Packages*

To run the included tests go to section: *Run the Tests*

Raspberry Pi OS

Testing platform is a Raspberry Pi 400 and the OS is the Raspberry Pi OS which runs on 64bit hardware but is a 32bit OS. The system Python 3 interpreter comes in version 3.7 (in 2021), but I will show in a later section how to install an additional newer Python version from the source code.

Install the build requirements, Matplotlib and the PyQt5 bindings from the distribution repository:

```
sudo apt install python3-pip python3-matplotlib python3-pyqt5
```

Installing Matplotlib and the PyQt5 bindings by *pip* from piwheels in the venv worked, but the packages showed errors at import, seems to be an packaging error in the required numpy package. PySide6 is the preferred Qt binding but wasn't available on Raspberry Pi OS at the time of writing this - PyQt5 is supported as fallback.

Create the venv with access to the system site-packages for using Matplotlib and the Qt bindings from the system installation:

```
cd ~
mkdir build
cd build
python3 -m venv --system-site-packages py37
source py37/bin/activate
```

Install *Cython* and *wheel* in the venv to get the C-extensions compiled:

```
pip3 install cython wheel
```

Clone the GitHub repository:

```
git clone https://github.com/mozman/ezdxf.git
```

Build and install ezdxf from source code:

```
cd ezdxf
pip3 install .
```

Check if the installation was successful:

```
python3 -m ezdxf -V
```

The output should look like this:

```
ezdxf 0.17.2b4 from /home/pi/src/py37/lib/python3.7/site-packages/ezdxf
Python version: 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0]
using C-extensions: yes
using Matplotlib: yes
```

To run the included tests go to section: *Run the Tests*

Manjaro on Raspberry Pi

Because the (very well working) Raspberry Pi OS is only a 32bit OS, I searched for a 64bit alternative like Ubuntu, which just switched to version 21.10 and always freezes at the installation process! So I tried Manjaro as rolling release, which I used prior in a virtual machine and wasn't really happy, because there is always something to update. Anyway the distribution looks really nice and has Python 3.9.9 installed.

Install build requirements and optional packages by the system packager *pacman*:

```
sudo pacman -S python-pip python-matplotlib python-pyqt5
```

Create and activate the venv:

```
cd ~
mkdir build
cd build
python3 -m venv --system-site-packages py39
source py39/bin/activate
```

The rest is the same procedure as for the *Raspberry Pi OS*:

```
pip3 install cython wheel
git clone https://github.com/mozman/ezdxf.git
cd ezdxf
pip3 install .
python3 -m ezdxf -V
```

To run the included tests go to section: *Run the Tests*

Ubuntu Server 21.10 on Raspberry Pi

I gave the Ubuntu Server 21.10 a chance after the desktop version failed to install by a nasty bug and it worked well. The distribution comes with Python 3.9.4 and after installing some requirements:

```
sudo apt install build-essential python3-pip python3.9-venv
```

The remaining process is like on *WSL & Ubuntu* except for the newer Python version. Installing Matplotlib by *pip* works as expected and is maybe useful even on a headless server OS to create SVG and PNG from DXF files. PySide6 is not available by *pip* and the installation of PyQt5 starts from the source code package which I stopped because this already didn't finished on Manjaro, but the installation of the PyQt5 bindings by *apt* works:

```
sudo apt install python3-pyqt5
```

Use the `--system-site-packages` option for creating the venv to get access to the PyQt5 package.

6.2.7 Install Optional Packages

Install the optional dependencies by *pip* only for *Windows 10* and *WSL & Ubuntu*, for *Raspberry Pi OS* and *Manjaro on Raspberry Pi* install these packages by the system packager:

```
pip3 install matplotlib PySide6
```

6.2.8 Run the Tests

This is the same procedure for all systems, assuming you are still in the build directory *build/ezdxf* and *ezdxf* is now installed in the venv.

Install the test dependencies and run the tests:

```
pip3 install pytest geomdl
python3 -m pytest tests integration_tests
```

6.2.9 Build Documentation

Assuming you are still in the build directory *build/ezdxf* of the previous section.

Install Sphinx:

```
pip3 install Sphinx sphinx-rtd-theme
```

Build the HTML documentation:

```
cd docs
make html
```

The output is located in *build/ezdxf/docs/build/html*.

6.2.10 Python from Source

Debian based systems have often very outdated software installed and sometimes there is no easy way to install a newer Python version. This is a brief summery how I installed Python 3.9.9 on the Raspberry Pi OS, for more information go to the source of the recipe: Real Python

Install build requirements:

```
sudo apt-get update
sudo apt-get upgrade

sudo apt-get install -y make build-essential libssl-dev zlib1g-dev \
   libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm \
   libncurses5-dev libncursesw5-dev xz-utils tk-dev
```

Make a build directory:

```
cd ~
mkdir build
cd build
```

Download and unpack the source code from Python.org, replace 3.9.9 by your desired version:

```
wget https://www.python.org/ftp/python/3.9.9/Python-3.9.9.tgz
tar -xvzf Python-3.9.9.tgz
cd Python-3.9.9/
```

Configure the build process, use a prefix to the directory where the interpreter should be installed:

```
./configure --prefix=/opt/python3.9.9 --enable-optimizations
```

Build & install the Python interpreter. The *-j* option simply tells *make* to split the building into parallel steps to speed up the compilation, my Raspberry Pi 400 has 4 cores so 4 seems to be a good choice:

```
make -j 4
sudo make install
```

The building time was ~25min and the new Python 3.9.9 interpreter is now installed as */opt/python3.9.9/bin/python3*.

At the time there were no system packages for Matplotlib and PyQt5 for this new Python version available, so there is no benefit of using the option `--system-site-packages` for building the venv:

```
cd ~/build
/opt/python3.9.9/bin/python3 -m venv py39
source py39/bin/activate
```

I have not tried to build Matplotlib and PyQt5 by myself and the installation by *pip* from piwheels did not work, in this case you don't get Matplotlib support for better font measuring and the *drawing* add-on will not work.

Proceed with the *ezdxf* installation from source as shown for the *Raspberry Pi OS*.

6.3 Usage for Beginners

This section shows the intended usage of the *ezdxf* package. This is just a brief overview for new *ezdxf* users, follow the provided links for more detailed information.

First import the package:

```
import ezdxf
```

6.3.1 Loading DXF Files

ezdxf supports loading ASCII and binary DXF documents from a file:

```
doc = ezdxf.readfile(filename)
```

or from a zip-file:

```
doc = ezdxf.readzip(zipfilename[, filename])
```

Which loads the DXF document *filename* from the zip-file *zipfilename* or the first DXF file in the zip-file if *filename* is absent.

It is also possible to read a DXF document from a stream by the `ezdxf.read()` function, but this is a more advanced feature, because this requires detection of the file encoding in advance.

This works well with DXF documents from trusted sources like AutoCAD or BricsCAD. For loading DXF documents with minor or major flaws use the `ezdxf.recover` module.

See also:

Documentation for `ezdxf.readfile()`, `ezdxf.readzip()` and `ezdxf.read()`, for more information about file management go to the *Document Management* section. For loading DXF documents with structural errors look at the `ezdxf.recover` module.

6.3.2 Layouts and Blocks

Layouts are containers for DXF entities like LINE or CIRCLE. The most important layout is the modelspace labeled as "Model" in CAD applications which represents the "world" work space. Paperspace layouts represents plottable sheets which contains often the framing and the tile block of a drawing and VIEWPORT entities as scaled and clipped "windows" into the modelspace.

The modelspace is always present and can not be deleted. The active paperspace is also always present in a new DXF document but can be deleted, in that case another paperspace layout gets the new active paperspace, but you can not delete the last paperspace layout.

Getting the modelspace of a DXF document:

```
msp = doc.modelspace()
```

Getting a paperspace layout by the name as shown in the tab of a CAD application:

```
psp = doc.paperspace("Layout1")
```

A block is just another kind of entity space, which can be inserted multiple times into other layouts and blocks by the INSERT entity also called block references, this is a very powerful and an important concept of the DXF format.

Getting a block layout by the block name:

```
blk = doc.blocks.get("NAME")
```

All these layouts have factory functions to create graphical DXF entities for their entity space, for more information about creating entities see section: *Create new DXF Entities*

6.3.3 Query DXF Entities

As said in the *Layouts and Blocks* section, all graphical DXF entities are stored in layouts, all these layouts can be iterated and do support the index operator e.g. `layout[-1]` returns the last entity.

The main difference between iteration and index access is, that iteration filters destroyed entities, but the index operator returns also destroyed entities until these entities are purged by `layout.purge()`, more about this topic in section: *Delete Entities*.

There are two advanced query methods: *query()* and *groupby()*.

Get all lines of layer `"MyLayer"`:

```
lines = msp.query('LINE[layer=="MyLayer"]')
```

This returns an *EntityQuery* container, which also provides the same `query()` and `groupby()` methods.

Get all lines categorized by a DXF attribute like color:

```
all_lines_by_color = msp.query("LINE").groupby("color")
lines_with_color_1 = all_lines_by_color.get(1, [])
```

The `groupby()` method returns a regular Python `dict` with colors as key and a regular Python `list` of entities as values (not an *EntityQuery* container).

See also:

For more information go to the *Tutorial for Getting Data from DXF Files*

6.3.4 Examine DXF Entities

Each DXF entity has a `dxf` namespace attribute, which stores the named DXF attributes, some entity attributes and assets are only available from Python properties or methods outside the `dxf` namespace like the vertices of the LWPOLYLINE entity. More information about the DXF attributes of each entity can found in the documentation of the *ezdxf.entities* module.

Get some basic DXF attributes:

```
layer = entity.dxf.layer  # default is "0"
color = entity.dxf.color  # default is 256 = BYLAYER
```

Most DXF attributes have a default value, which will be returned if the DXF attribute is not present, for DXF attributes without a default value you can check if the attribute really exist:

```
entity.dxf.hasattr("true_color")
```

or use the `get()` method and provide a default value:

```
entity.dxf.get("true_color", 0)
```

See also:

- *Common graphical DXF attributes*
- Helper class *ezdxf.gfxattribs.GfxAttribs* for building DXF attribute dictionaries.

6.3.5 Create a New DXF File

Create new document for the latest supported DXF version:

```
doc = ezdxf.new()
```

Create a new DXF document for a specific DXF version, e.g. for DXF R12:

```
doc = ezdxf.new("R12")
```

The *ezdxf.new()* function can create some standard resources, such as linetypes and text styles, by setting the argument *setup* to `True`:

```
doc = ezdxf.new(setup=True)
```

See also:

- *Tutorial for Creating DXF Drawings*
- Documentation for *ezdxf.new()*, for more information about file management go to the *Document Management* section.

6.3.6 Create New DXF Entities

The factory methods for creating new graphical DXF entities are located in the *BaseLayout* class and these factory methods are available for all entity containers:

- *Modelspace*
- *Paperspace*
- *BlockLayout*

The usage is simple:

```
msp = doc.modelspace()
msp.add_line((0, 0), (1, 0), dxfattribs={"layer": "MyLayer"})
```

A few important/required DXF attributes are explicit method arguments, most additional DXF attributes are gives as a regular Python `dict` object by the keyword only argument `dxfattribs`. The supported DXF attributes can be found in the documentation of the *ezdxf.entities* module.

Warning: Do not instantiate DXF entities by yourself and add them to layouts, always use the provided factory methods to create new graphical entities, this is the intended way to use *ezdxf*.

See also:

- *Thematic Index of Layout Factory Methods*
- *Tutorial for Creating DXF Drawings*
- *Tutorial for Simple DXF Entities*
- *Tutorial for LWPolyline*
- *Tutorial for Text*
- *Tutorial for MText and MTextEditor*
- *Tutorial for Hatch*

6.3.7 Saving DXF Files

Save the DXF document with a new name:

```
doc.saveas("new_name.dxf")
```

or with the same name as loaded:

```
doc.save()
```

See also:

Documentation for *ezdxf.document.Drawing.save()* and *ezdxf.document.Drawing.saveas()*, for more information about file management go to the *Document Management* section.

6.3.8 Create New Blocks

The block definitions of a DXF document are managed by the *BlocksSection* object:

```
my_block = doc.blocks.new("MyBlock")
```

See also:

Tutorial for Blocks

6.3.9 Create Block References

A block reference is just another DXF entity called INSERT. The *Insert* entity is created by the factory method: *add_blockref()*:

```
msp.add_blockref("MyBlock", (0, 0))
```

See also:

See *Tutorial for Blocks* for more advanced features like using *Attrib* entities.

6.3.10 Create New Layers

A layer is not an entity container, a layer is just another DXF attribute stored in the entity and the entity can inherit some properties from this *Layer* object. Layer objects are stored in the layer table which is available as attribute `doc.layers`.

You can create your own layers:

```
my_layer = doc.layer.add("MyLayer")
```

The layer object also controls the visibility of entities which references this layer, the on/off state of the layer is unfortunately stored as positive or negative color value which make the raw DXF attribute of layers useless, to change the color of a layer use the property `Layer.color`

```
my_layer.color = 1
```

To change the state of a layer use the provided methods of the *Layer* object, like *on()*, *off()*, *freeze()* or *thaw()*:

```
my_layer.off()
```

See also:

Layers

6.3.11 Delete Entities

The safest way to delete entities is to delete the entity from the layout containing that entity:

```
line = msp.add_line((0, 0), (1, 0))
msp.delete_entity(line)
```

This removes the entity immediately from the layout and destroys the entity. The property *is_alive* returns `False` for a destroyed entity and all Python attributes are deleted, so `line.dxf.color` will raise an `AttributeError` exception, because `line` does not have a *dxf* attribute anymore.

Ezdxf also supports manually destruction of entities by calling the method `destroy()`:

```
line.destroy()
```

Manually destroyed entities are not removed immediately from entities containers like `Modelspace` or `EntityQuery`, but iterating such a container will filter destroyed entities automatically, so a `for e in msp: ...` loop will never yield destroyed entities. The index operator and the `len()` function do **not** filter deleted entities, to avoid getting deleted entities call the `purge()` method of the container manually to remove deleted entities.

6.3.12 Further Information

- *Reference*

6.4 Basic Concepts

The Basic Concepts section teach the intended meaning of DXF attributes and structures without teaching the application of this information or the specific implementation by *ezdxf*, if you are looking for more information about the *ezdxf* internals look at the *Reference* section or if you want to learn how to use *ezdxf* go to the *Tutorials* section and for the solution of specific problems go to the *Howto* section.

6.4.1 What is DXF?

The common assumption is also the cite of Wikipedia:

> AutoCAD DXF (Drawing eXchange Format) is a CAD data file format developed by Autodesk for enabling data interoperability between AutoCAD and **other** applications.
>
> DXF was originally introduced in December 1982 as part of AutoCAD 1.0, and was intended to provide an exact representation of the data in the AutoCAD native file format, DWG (Drawing). For many years Autodesk did not publish specifications making correct imports of DXF files difficult. Autodesk now publishes the DXF specifications online.

The more precise cite from the DXF reference itself:

> The DXF™ format is a tagged data representation of all the information contained in an AutoCAD® drawing file. Tagged data means that each data element in the file is preceded by an integer number that is called a group code. A group code's value indicates what type of data element follows. This value also indicates the meaning of a data element for a given object (or record) type. Virtually all user-specified information in a drawing file can be represented in DXF format.

No mention of interoperability between AutoCAD and **other** applications.

In reality the DXF format was designed to ensure AutoCAD cross-platform compatibility in the early days when different hardware platforms with different binary data formats were used. The name DXF (Drawing eXchange Format) may suggest an universal exchange format, but it is not. It is based on the infrastructure installed by Autodesk products (fonts) and the implementation details of AutoCAD (MTEXT) or on licensed third party technologies (embedded ACIS entities).

For more information about the AutoCAD history see the document: The Autodesk File - Bits of History, Words of Experience by *John Walker*, founder of *Autodesk, Inc.* and co-author of *AutoCAD*.

DXF Reference Quality

The DXF reference is by far no specification nor a standard like the W3C standard for SVG or the ISO standard for PDF.

The reference describes many but not all DXF entities and some basic concepts like the tag structure or the arbitrary axis algorithm. But the existing documentation (reference) is incomplete and partly misleading or wrong. Also missing from the reference are some important parts like the complex relationship between the entities to create higher order structures like block definitions, layouts (model space & paper space) or dynamic blocks to name a few.

Reliable CAD Applications

Because of the suboptimal quality of the DXF reference not all DXF viewers, creators or processors are of equal quality. I consider a CAD application as a *reliable CAD application* when the application creates valid DXF documents in the meaning and interpretation of Autodesk and a reliable DXF viewer when the result matches in most parts the result of the free Trueview viewer provided by Autodesk.

These are some applications which do fit the criteria of a reliable CAD application:

- AutoCAD and Trueview
- CAD applications based on the OpenDesignAlliance (ODA) SDK, see also ODA on wikipedia, even Autodesk is a corporate member, see their blog post from 22 Sep 2020 at adsknews but only to use the ODA IFC tools and not to improve the DWG/DXF compatibility
- BricsCAD (ODA based)
- GstarCAD (ODA based)
- ZWCAD (ODA based)

Unfortunately, I cannot recommend any open source applications because everyone I know has serious shortcomings, at least as a DXF viewer, and I don't trust them as a DXF creator either. To be clear, not even *ezdxf* (which is not a CAD application) is a *reliable* library in this sense - it just keeps getting better, but is far from *reliable*.

Hint: Please do not submit bug reports based on the use of LibreCAD or QCAD, these applications are in no way reliable regarding the DXF format and I will not waste my time on them.

6.4.2 DXF Entities and Objects

DXF entities are objects that make up the design data stored in a DXF file.

Graphical Entities

Graphical entities are visible objects stored in blocks, modelspace- or paperspace layouts. They represent the various shapes, lines, and other elements that make up a 2D or 3D design.

Some common types of DXF entities include:

- LINE and POLYLINE: These are the basic building blocks of a DXF file. They represent straight and curved lines.
- CIRCLE and ARC: These entities represent circles and portions of circles, respectively.
- TEXT and MTEXT: DXF files can also contain text entities, which can be used to label parts of the design or provide other information.
- HATCH: DXF files can also include hatch patterns, which are used to fill in areas with a specific pattern or texture.
- DIMENSION: DXF files can also contain dimension entities, which provide precise measurements of the various elements in a design.
- INSERT: A block is a group of entities that can be inserted into a design multiple times by the INSERT entity, making it a useful way to reuse elements of a design.

These entities are defined using specific codes and values in the DXF file format, and they can be created and manipulated by *ezdxf*.

Objects

DXF objects are non-graphical entities and have no visual representation, they store administrative data, paperspace layout definitions, style definitions for multiple entity types, custom data and objects. The OBJECTS section in DXF files serves as a container for these non-graphical objects.

Some common DXF types of DXF objects include:

- DICTIONARY: A dictionary object consists of a series of name-value pairs, where the name is a string that identifies a specific object within the dictionary, and the value is a reference to that object. The objects themselves can be any type of DXF entity or custom object defined in the DXF file.
- XRECORD entities are used to store custom application data in a DXF file.
- the LAYOUT entity is a DXF entity that represents a single paper space layout in a DXF file. Paper space is the area in a CAD drawing that represents the sheet of paper or other physical media on which the design will be plotted or printed.
- MATERIAL, MLINESTYLE, MLEADERSTYLE definitions stored in certain DICTIONARY objects.
- A GROUP entity contains a list of handles that refer to other DXF entities in the drawing. The entities in the group can be of any type, including entities from the model space or paper space layouts.

TagStorage

The *ezdxf* package supports many but not all entity types, all these unsupported types are stored as `TagStorage` instances to preserve their data when exporting the edited DXF content by *ezdxf*.

Access Entity Attributes

All DXF attributes are stored in the entity namespace attribute `dxf`.

```
print(entity.dxf.layer)
```

Some attributes are mandatory others are optional in most cases a reasonable values will be returned as default value if the attribute is missing.

See also:

Tutorial for Getting Data from DXF Files

Where to Look for Entities

The DXF document has an entity database where all entities which have a handle are stored in a (key, value) storage. The `query()` method is often the easiest way to request data:

```
for text in doc.entitydb.query("TEXT"):
    print(text.dxf.text)
```

See also:

- *ezdxf.query* module
- *ezdxf.entitydb* module

Graphical entities are stored in blocks, the modelspace or paperspace layouts.

- The `doc.modelspace()` function returns the *Modelspace* instance
- The `doc.paperspace()` returns a *Paperspace* instance
- The `doc.blocks` attribute provides access to the *BlocksSection*

The *query()* method of the *Drawing* class which represents the DXF document, runs the query on all layouts and block definitions.

Non-graphical entities are stored in the OBJECTS section:

- The `doc.objects` attribute provides access to the *ObjectsSection*.

Resource definitions like *Layer*, *Linetype* or *Textstyle* are stored in resource tables:

- `doc.layers`: the *LayerTable*
- `doc.linetypes`: the *LinetypeTable*
- `doc.styles`: the *TextstyleTable*
- `doc.dimstyles`: the *DimStyleTable*

Important: A layer assignment is just an attribute of a DXF entity, it's not an entity container!

See also:

- Basic concept of the *Modelspace*
- Basic concept of *Paperspace* layouts
- Basic concept of *Blocks*
- *Tutorial for Getting Data from DXF Files*

How to Create Entities

The recommended way to create new DXF entities is to use the factory methods of layouts and blocks to create entities and add them to the entity space automatically.

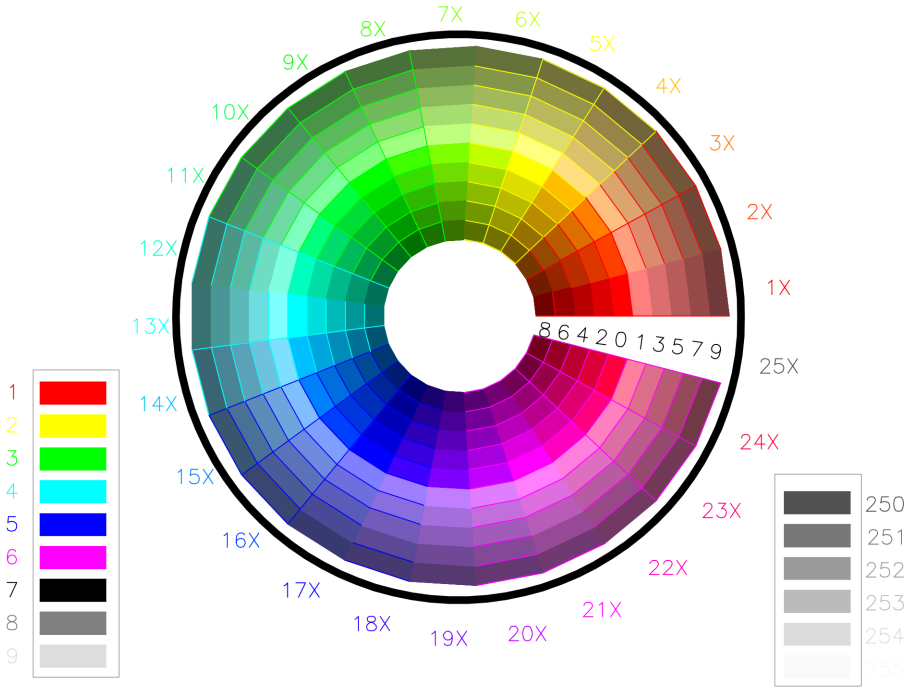
See also:

- *Thematic Index of Layout Factory Methods*
- Reference of the `BaseLayout` class
- *Tutorial for Simple DXF Entities*

6.4.3 AutoCAD Color Index (ACI)

The `color` attribute represents an *ACI* (AutoCAD Color Index). AutoCAD and many other *CAD* application provides a default color table, but pen table would be the more correct term. Each ACI entry defines the color value, the line weight and some other attributes to use for the pen. This pen table can be edited by the user or loaded from an *CTB* or *STB* file. *Ezdxf* provides functions to create (`new()`) or modify (`ezdxf.acadctb.load()`) plot styles files.

DXF R12 and prior do not preserve the layout of a drawing very well, because of the lack of a standard color table and missing DXF structures to define these color tables in the DXF file. If a CAD user redefines an ACI color entry in a CAD application and does not provide this *CTB* or *STB* file, you can not know what color or lineweight was used intentionally. This got better in later DXF versions by supporting additional DXF attributes like `lineweight` and `true_color` which can define these attributes by distinct values.



See also:

- *Plot Style Files (CTB/STB)*
- `ezdxf.colors`
- *Tutorial for Common Graphical Attributes*
- Autodesk Knowledge Network: About Setting the Color of Objects
- BricsCAD Help Center: Entity Color

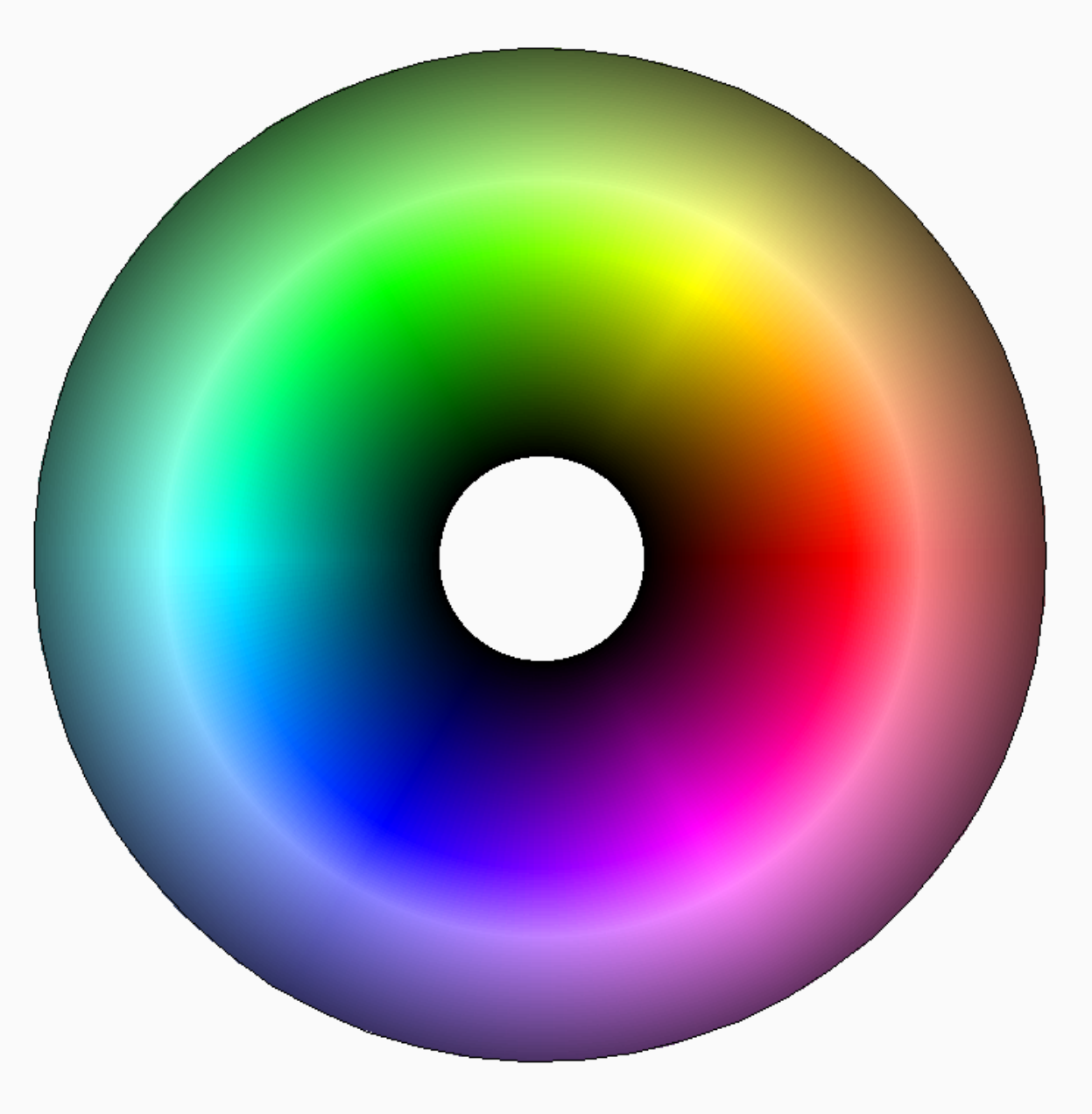
6.4.4 True Color

The support for true color was added to the DXF file format in revision R2004. The true color value has three components red, green and blue in the range from 0 to 255 and is stored as a 24-bit value in the DXF namespace as `true_color` attribute and looks like this `0xRRGGBB` as hex value. For a more easy usage all graphical entities support the *rgb* property to get and set the true color as (r, g, b) tuples where the components must be in the range from 0 to 255.

```
import ezdxf

doc = ezdxf.new()
msp = doc.modelspace()
line = msp.add_line((0, 0), (10, 0))
line.rgb = (255, 128, 32)
```

The true color value has higher precedence than the *AutoCAD Color Index (ACI)* value, if the attributes `color` and the `true_color` are present the entity will be rendered with the true color value.

The true color value has the advantage that it defines the color absolutely and unambiguously, no unexpected overwriting is possible. The representation of the color is fixed and only depends on the calibration of the output medium:



See also:

- `ezdxf.colors`
- *Tutorial for Common Graphical Attributes*
- Autodesk Knowledge Network: About Setting the Color of Objects
- BricsCAD Help Center: Entity Color

6.4.5 Transparency

The support for transparency was added to the DXF file format in revision R2004. The raw transparency value stored as 32 bit value in the DXF namespace as `transparency` attribute, has a range from 0 to 255 where 0 is fully transparent and 255 if opaque and has the top byte set to `0x02`. For a more easy usage all graphical entities support the *transparency* property to get and set the transparency as float value in the range frem 0.0 to 1.0 where 0.0 is opaque and 1.0 is fully transparent. The transparency value can be set explicit in the entity, by layer or by block.

```
import ezdxf

doc = ezdxf.new()
msp = doc.modelspace()
line = msp.add_line((0, 0), (10, 0))
line.transparency = 0.5
```

See also:

- *ezdxf.colors*
- *Tutorial for Common Graphical Attributes*
- Autodesk Knowledge Network: About Making Objects Transparent
- BricsCAD Help Center: Entity Transparency

6.4.6 Layers

Every object has a layer as one of its properties. You may be familiar with layers - independent drawing spaces that stack on top of each other to create an overall image - from using drawing programs. Most CAD programs use layers as the primary organizing principle for all the objects that you draw. You use layers to organize objects into logical groups of things that belong together; for example, walls, furniture, and text notes usually belong on three separate layers, for a couple of reasons:

- Layers give you a way to turn groups of objects on and off - both on the screen and on the plot.
- Layers provide the most efficient way of controlling object color and linetype

Create a layer table entry *Layer* by `Drawing.layers.add()`, assign the layer properties such as color and linetype. Then assign those layers to other DXF entities by setting the DXF attribute *layer* to the layer name as string.

The DXF format do not require a layer table entry for a layer. A layer without a layer table entry has the default linetype `'Continuous'`, a default color of `7` and a lineweight of `-3` which represents the default lineweight of 0.25mm in most circumstances.

Layer Properties

The advantage of assigning properties to a layer is that entities can inherit this properties from the layer by using the string `'BYLAYER'` as linetype string, `256` as color or `-1` as lineweight, all these values are the default values for new entities. DXF version R2004 and later also support inheriting *true_color* and *transparency* attributes from a layer.

Layer Status

The layer status is important for the visibility and the ability to select and edit DXF entities on that layer in CAD applications. *Ezdxf* does not care about the visual representation and works at the level of entity spaces and the entity database and therefore all the layer states documented below are ignored by *ezdxf*. This means if you iterate an entity space like the modelspace or the entity database you will get all entities from that entity space regardless the layer status.

- ON: the layer is visible, entities on that layer are visible, selectable and editable
- OFF: the layer is not visible, entities on that layer are not visible, not selectable and not editable
- FROZEN: the layer is not visible, entities on that layer are not visible, not selectable and not editable, very similar to the OFF status but layers can be frozen individually in VIEWPORTS and freezing layers may speed up some commands in CAD applications like ZOOM, PAN or REGEN.
- LOCKED: the layer is visible, entities on that layer are visible but not selectable and not editable

Deleting Layers

Deleting a layer is not as simple as it might seem, especially if you are used to use a CAD application like AutoCAD. There is no directory of locations where layers can be used and references to layers can occur even in third-party data. Deleting the layer table entry removes only the default attributes of that layer and does not delete any layer references automatically. And because a layer can exist without a layer table entry, the layer exist as long as at least one layer reference to the layer exist.

Renaming Layers

Renaming a layer is also problematic because the DXF format stores the layer references in most cases as text strings, so renaming the layer table entry just creates a new layer and all entities which still have a reference to the old layer now inherit their attributes from an undefined layer table entry with default settings.

Viewport Overrides

Most of the layer properties can be overriden for each `Viewport` entity individually and this overrides are stored in layer table entry referenced by the handle of the VIEWPORT entity. In contrast the frozen status of layers is store in the VIEWPORT entity.

See also:

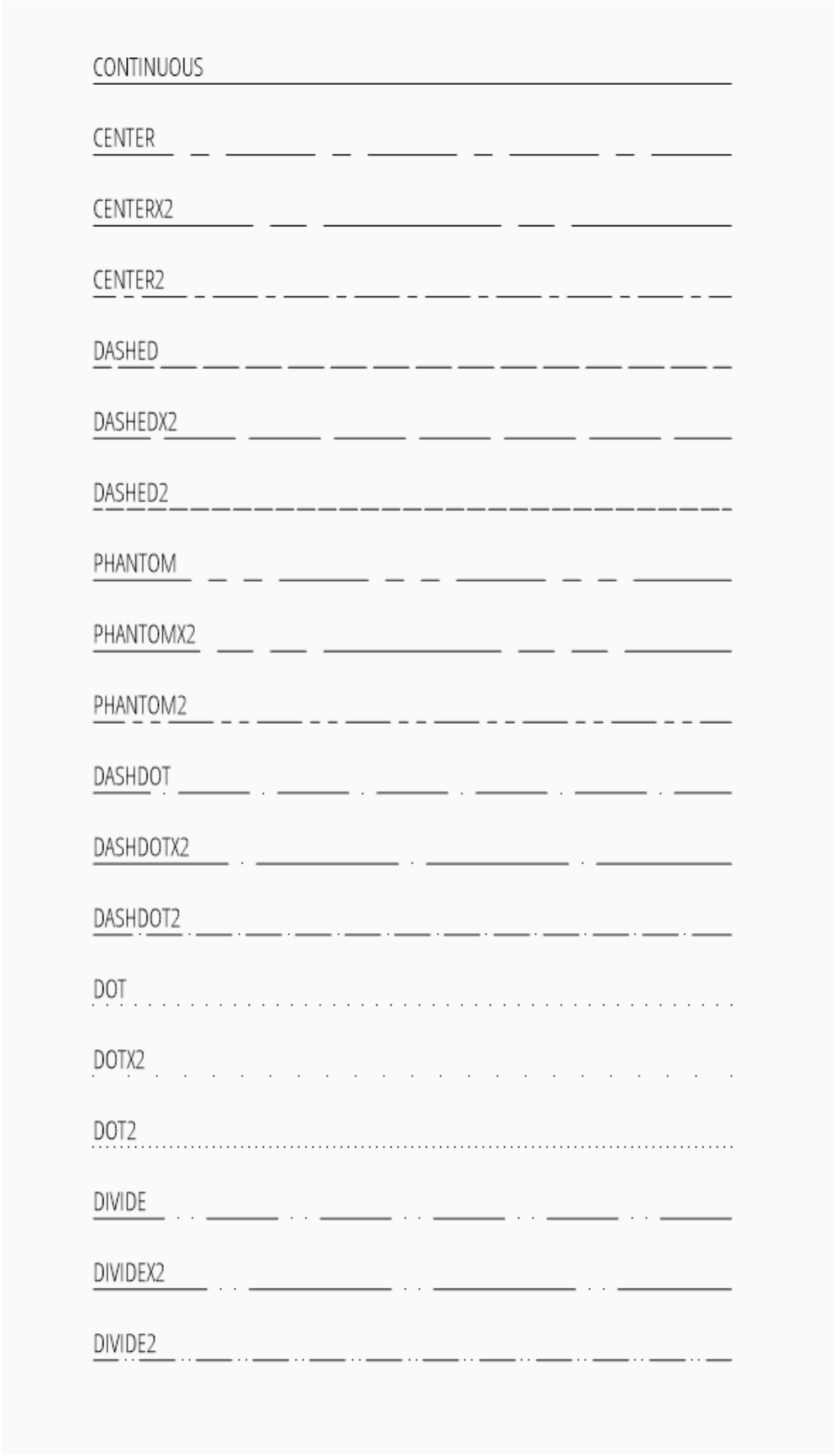
- *Tutorial for Layers*
- *Tutorial for Viewports in Paperspace*
- Autodesk Knowledge Network: About Layers
- BricsCAD Help Center: Working with Layers

6.4.7 Linetypes

The *linetype* defines the rendering pattern of linear graphical entities like LINE, ARC, CIRCLE and so on. The linetype of an entity can be specified by the DXF attribute `linetype`, this can be an explicit named linetype or the entity can inherit its linetype from the assigned layer by setting `linetype` to `'BYLAYER'`, which is also the default value. `CONTINUOUS` is the default linetype for layers with an unspecified linetype.

Ezdxf creates several standard linetypes, if the argument *setup* is `True` when calling *new()*, this simple linetypes are supported by all DXF versions:

```
doc = ezdxf.new('R2007', setup=True)
```


CONTINUOUS
CENTER
CENTERX2
CENTER2
DASHED
DASHEDX2
DASHED2
PHANTOM
PHANTOMX2
PHANTOM2
DASHDOT
DASHDOTX2
DASHDOT2
DOT
DOTX2
DOT2
DIVIDE
DIVIDEX2
DIVIDE2

In DXF R13 Autodesk introduced complex linetypes which can contain text or shapes.

See also:

- *Tutorial for Common Graphical Attributes*
- *Tutorial for Creating Linetype Pattern*
- Autodesk Knowledge Network: About Linetypes
- BricsCAD Help Center: Entity Linetype

Linetype Scaling

Global linetype scaling can be changed by setting the header variable `doc.header['$LTSCALE'] = 2`, which stretches the line pattern by factor 2.

The linetype scaling for a single entity can be set by the DXF attribute *`ltscale`*, which is supported since DXF R2000.

6.4.8 Lineweights

The *`lineweight`* attribute represents the lineweight as integer value in millimeters * 100, e.g. 0.25mm = 25, independently from the unit system used in the DXF document. The `lineweight` attribute is supported by DXF R2000 and newer.

Only certain values are valid, they are stored in `ezdxf.lldxf.const.VALID_DXF_LINEWEIGHTS`: 0, 5, 9, 13, 15, 18, 20, 25, 30, 35, 40, 50, 53, 60, 70, 80, 90, 100, 106, 120, 140, 158, 200, 211.

Values < 0 have a special meaning and can be imported as constants from *`ezdxf.lldxf.const`*

-1	LINEWEIGHT_BYLAYER
-2	LINEWEIGHT_BYBLOCK
-3	LINEWEIGHT_DEFAULT

The validator function: `ezdxf.lldxf.validator.is_valid_lineweight()` returns `True` for valid lineweight values otherwise `False`.

Sample script which shows all valid lineweights: valid_lineweights.dxf

You have to enable the option to show lineweights in your CAD application or viewer to see the effect on screen, which is disabled by default, the same has to be done in the page setup options for plotting lineweights.

Setting the HEADER variable `$LWDISPLAY` to 1, activates support for displaying lineweights on screen:

```
# activate on screen lineweight display
doc.header["$LWDISPLAY"] = 1
```


Lineweight: 2.11
Lineweight: 2.00
Lineweight: 1.58
Lineweight: 1.40
Lineweight: 1.20
Lineweight: 1.06
Lineweight: 1.00
Lineweight: 0.90
Lineweight: 0.80
Lineweight: 0.70
Lineweight: 0.60
Lineweight: 0.53
Lineweight: 0.50
Lineweight: 0.40
Lineweight: 0.35
Lineweight: 0.30
Lineweight: 0.25
Lineweight: 0.20
Lineweight: 0.18
Lineweight: 0.15
Lineweight: 0.13
Lineweight: 0.10
Lineweight: 0.09
Lineweight: 0.05
Lineweight: 0.00

The lineweight value can be overridden by *CTB* or *STB* files.

See also:

- Autodesk Knowledge Network: About Lineweights
- BricsCAD Help Center: Entity Lineweight

6.4.9 Coordinate Systems

AutoLISP Reference to Coordinate Systems provided by Autodesk.

To brush up you knowledge about vectors, watch the YouTube tutorials of 3Blue1Brown about Linear Algebra.

WCS

World coordinate system - the reference coordinate system. All other coordinate systems are defined relative to the WCS, which never changes. Values measured relative to the WCS are stable across changes to other coordinate systems.

UCS

User coordinate system - the working coordinate system defined by the user to make drawing tasks easier. All points passed to AutoCAD commands, including those returned from AutoLISP routines and external functions, are points in the current UCS. As far as I know, all coordinates stored in DXF files are always WCS or OCS never UCS.

User defined coordinate systems are not just helpful for interactive CAD, therefore *ezdxf* provides a converter class *UCS* to translate coordinates from UCS into WCS and vice versa, but always remember: store only WCS or OCS coordinates in DXF files, because there is no method to determine which UCS was active or used to create UCS coordinates.

See also:

- Table entry `UCS`
- *ezdxf.math.UCS* - converter between WCS and UCS

OCS

Object coordinate system are coordinates relative to the object itself. The main goal of OCS is to place 2D elements in 3D space and the OCS is defined by the extrusion vector of the entity. As long the extrusion vector is (0, 0, 1) (the WCS z-axis) the OCS is coincident to the WCS, which means the OCS coordinates are equal to the WCS coordinates, most of the time this is true for 2D entities.

OCS entities: ARC, CIRCLE, TEXT, LWPOLYLINE, HATCH, SOLID, TRACE, INSERT, IMAGE

Because *ezdxf* is just an interface to DXF, it does not automatically convert OCS into WCS, this is the domain of the user/application. These lines convert the center of a 3D circle from OCS to WCS:

```
ocs = circle.ocs()
wcs_center = ocs.to_wcs(circle.dxf.center)
```

See also:

- *Object Coordinate System (OCS)* - deeper insights into OCS
- *ezdxf.math.OCS* - converter between WCS and OCS

DCS

Display coordinate system - the coordinate system into which objects are transformed before they are displayed. The origin of the DCS is the point stored in the AutoCAD system variable TARGET, and its z-axis is the viewing direction. In other words, a viewport is always a plan view of its DCS. These coordinates can be used to determine where something will be displayed to the AutoCAD user. *Ezdxf* does not use or support DCS in any way.

6.4.10 Object Coordinate System (OCS)

- DXF Reference for OCS provided by Autodesk.

The points associated with each entity are expressed in terms of the entity's own object coordinate system (OCS). The OCS was referred to as ECS in previous releases of AutoCAD.

With OCS, the only additional information needed to describe the entity's position in 3D space is the 3D vector describing the z-axis of the OCS (often referenced as extrusion vector), and the elevation value, which is the distance of the entity xy-plane to the WCS/OCS origin.

For a given z-axis (extrusion) direction, there are an infinite number of coordinate systems, defined by translating the origin in 3D space and by rotating the x- and y-axis around the z-axis. However, for the same z-axis direction, there is only one OCS. It has the following properties:

- Its origin coincides with the WCS origin.
- The orientation of the x- and y-axis within the xy-plane are calculated in an arbitrary but consistent manner. AutoCAD performs this calculation using the arbitrary axis algorithm (see below).
- Because of the *Arbitrary Axis Algorithm* the OCS can only represent a **right-handed** coordinate system!

The following entities do not lie in a particular plane. All points are expressed in world coordinates. Of these entities, only lines and points can be extruded. Their extrusion direction can differ from the world z-axis.

- *Line*
- *Point*
- 3DFace
- *Polyline* (3D)
- *Vertex* (3D)
- *Polymesh*
- *Polyface*
- *Viewport*

These entities are planar in nature. All points are expressed in object coordinates. All of these entities can be extruded. Their extrusion direction can differ from the world z-axis.

- *Circle*
- *Arc*
- *Solid*
- *Trace*
- *Text*
- *Attrib*
- Attdef

- *Shape*
- *Insert*
- *Polyline* (2D)
- *Vertex* (2D)
- *LWPolyline*
- *Hatch*
- *Image*

Some of a *Dimension*'s points are expressed in WCS and some in OCS.

Elevation

Elevation group code 38:

Exists only in output from versions prior to R11. Otherwise, Z coordinates are supplied as part of each of the entity's defining points.

Arbitrary Axis Algorithm

- DXF Reference for Arbitrary Axis Algorithm provided by Autodesk.

The arbitrary axis algorithm is used by AutoCAD internally to implement the arbitrary but consistent generation of object coordinate systems for all entities that use object coordinates.

Given a unit-length vector to be used as the z-axis of a coordinate system, the arbitrary axis algorithm generates a corresponding x-axis for the coordinate system. The y-axis follows by application of the **right-hand** rule.

We are looking for the arbitrary x- and y-axis to go with the normal Az (the arbitrary z-axis). They will be called Ax and Ay (using *Vec3*):

```
Az = Vec3(entity.dxf.extrusion).normalize()  # normal (extrusion) vector
if (abs(Az.x) < 1/64.) and (abs(Az.y) < 1/64.):
     Ax = Vec3(0, 1, 0).cross(Az).normalize()  # the cross-product operator
else:
     Ax = Vec3(0, 0, 1).cross(Az).normalize()  # the cross-product operator
Ay = Az.cross(Ax).normalize()
```

WCS to OCS

```
def wcs_to_ocs(point):
    px, py, pz = Vec3(point)  # point in WCS
    x = px * Ax.x + py * Ax.y + pz * Ax.z
    y = px * Ay.x + py * Ay.y + pz * Ay.z
    z = px * Az.x + py * Az.y + pz * Az.z
    return Vec3(x, y, z)
```

OCS to WCS

```
Wx = wcs_to_ocs((1, 0, 0))
Wy = wcs_to_ocs((0, 1, 0))
Wz = wcs_to_ocs((0, 0, 1))

def ocs_to_wcs(point):
    px, py, pz = Vec3(point)  # point in OCS
    x = px * Wx.x + py * Wy.x + pz * Wz.x
    y = px * Wx.y + py * Wy.y + pz * Wz.y
    z = px * Wx.z + py * Wy.z + pz * Wz.z
    return Vec3(x, y, z)
```

6.4.11 DXF Units

The DXF reference has no explicit information how to handle units in DXF, any information in this section is based on experiments with BricsCAD and may differ in other CAD applications, BricsCAD tries to be as compatible with AutoCAD as possible. Therefore, this information should also apply to AutoCAD.

Please open an issue on github if you have any corrections or additional information about this topic.

Length Units

Any length or coordinate value in DXF is unitless in the first place, there is no unit information attached to the value. The unit information comes from the context where a DXF entity is used. The document/modelspace get the unit information from the header variable $INSUNITS, paperspace and block layouts get their unit information from the attribute *units*. The modelspace object has also a `units` property, but this value do not represent the modelspace units, this value is always set to 0 "unitless".

Get and set document/modelspace units as enum by the *Drawing* property `units`:

```
import ezdxf
from ezdxf import units

doc = ezdxf.new()
# Set centimeter as document/modelspace units
doc.units = units.CM
# which is a shortcut (including validation) for
doc.header['$INSUNITS'] = units.CM
```

Block Units

As said each block definition can have independent units, but there is no implicit unit conversion applied, not in CAD applications and not in *ezdxf*.

When inserting a block reference (INSERT) into the modelspace or another block layout with different units, the scaling factor between these units **must** be applied explicit as DXF attributes (`xscale`, ...) of the *Insert* entity, e.g. modelspace in meters and block in centimeters, x-, y- and z-scaling has to be 0.01:

```
doc.units = units.M
my_block = doc.blocks.new('MYBLOCK')
my_block.units = units.CM
block_ref = msp.add_block_ref('MYBLOCK')
```

(continues on next page)

(continued from previous page)

```
# Set uniform scaling for x-, y- and z-axis
block_ref.set_scale(0.01)
```

Use helper function *conversion_factor()* to calculate the scaling factor between units:

```
factor = units.conversion_factor(doc.units, my_block.units)
# factor = 100 for 1m is 100cm
# scaling factor = 1 / factor
block_ref.set_scale(1.0/factor)
```

Hint: It is never a good idea to use different measurement system in one project, ask the NASA about their Mars Climate Orbiter from 1999. The same applies for units of the same measurement system, just use one unit like meters or inches.

Angle Units

Angles are always in degrees (360 deg = full circle) in counter-clockwise orientation, unless stated explicit otherwise.

Display Format

How values are shown in the CAD GUI is controlled by the header variables $LUNITS and $AUNITS, but this has no meaning for values stored in DXF files.

$INSUNITS

The most important setting is the header variable $INSUNITS, this variable defines the drawing units for the modelspace and therefore for the DXF document if no further settings are applied.

The modelspace LAYOUT entity has a property *units* as any layout like object, but it seem to have no meaning for the modelspace, BricsCAD set this property always to 0, which means unitless.

The most common units are 6 for meters and 1 for inches.

```
doc.header['$INSUNITS'] = 6
```

0	Unitless
1	Inches, `units.IN`
2	Feet, `units.FT`
3	Miles, `units.MI`
4	Millimeters, `units.MM`
5	Centimeters, `units.CM`
6	Meters, `units.M`
7	Kilometers, `units.KM`
8	Microinches
9	Mils
10	Yards, `units.YD`
11	Angstroms
12	Nanometers
13	Microns
14	Decimeters, `units.DM`
15	Decameters
16	Hectometers
17	Gigameters
18	Astronomical units
19	Light years
20	Parsecs
21	US Survey Feet
22	US Survey Inch
23	US Survey Yard
24	US Survey Mile

See also enumeration *ezdxf.enums.InsertUnits*.

$MEASUREMENT

The header variable $MEASUREMENT controls whether the current drawing uses imperial or metric hatch pattern and linetype files:

This setting is independent from $INSUNITS, it is possible to set the drawing units to inch and use metric linetypes and hatch pattern.

In BricsCAD the base scaling of linetypes and hatch pattern is defined by the $MEASUREMENT value, the value of $INSUNITS is ignored.

```
doc.header['$MEASUREMENT'] = 1
```

0	English
1	Metric

See also enumeration *ezdxf.enums.Measurement*

$LUNITS

The header variable $LUNITS defines how CAD applications display linear values in the GUI and has no meaning for *ezdxf*:

```
doc.header['$LUNITS'] = 2
```

1	Scientific
2	Decimal (default)
3	Engineering
4	Architectural
5	Fractional

See also enumeration *ezdxf.enums.LengthUnits*

$AUNITS

The header variable $AUNITS defines how CAD applications display angular values in the GUI and has no meaning for *ezdxf*, DXF angles are always stored as degrees in counter-clockwise orientation, unless stated explicit otherwise:

```
doc.header['$AUNITS'] = 0
```

0	Decimal degrees
1	Degrees/minutes/seconds
2	Grad
3	Radians

See also enumeration *ezdxf.enums.AngularUnits*

Helper Tools

ezdxf.units.**conversion_factor**(*source_units:* InsertUnits, *target_units:* InsertUnits) → float

Returns the conversion factor to represent *source_units* in *target_units*.

E.g. millimeter in centimeter `conversion_factor(MM, CM)` returns 0.1, because 1 mm = 0.1 cm

ezdxf.units.**unit_name**(*enum: int*) → str

Returns the name of the unit enum.

ezdxf.units.**angle_unit_name**(*enum: int*) → str

Returns the name of the angle unit enum.

6.4.12 Modelspace

The modelspace contains the "real" world representation of the drawing subjects in real world units and is displayed in the tab called "Model" in CAD applications. The modelspace is always present and can't be deleted.

The modelspace object is acquired by the method `modelspace()` of the `Drawing` class and new entities should be added to the modelspace by factory methods: *Thematic Index of Layout Factory Methods*.

This is a common idiom for creating a new document and acquiring the modelspace:

```
import ezdxf

doc = ezdxf.new()
msp = doc.modelspace()
```

The modelspace can have one or more rectangular areas called modelspace viewports. The modelspace viewports can be used for displaying different views of the modelspace from different locations of the modelspace or viewing directions. It is important to know that modelspace viewports (`VPort`) are not the same as paperspace viewport entities (`Viewport`).

See also:

- Reference of class `Modelspace`
- *Thematic Index of Layout Factory Methods*
- Example for usage of modelspace viewports: tiled_window_setup.py

6.4.13 Paperspace

A paperspace layout is where the modelspace drawing content is assembled and organized for 2D output, such as printing on a sheet of paper, or as a digital document, such as a PDF file.

Each DXF document can have one or more paperspace layouts but the DXF version R12 supports only one paperspace layout and it is not recommended to rely on paperspace layouts in DXF version R12.

Graphical entities can be added to the paperspace by factory methods: *Thematic Index of Layout Factory Methods*. Views or "windows" to the modelspace are added as `Viewport` entities, each viewport displays a region of the modelspace and can have an individual scaling factor, rotation angle, clipping path, view direction or overridden layer attributes.

See also:

- Reference of class `Paperspace`
- *Thematic Index of Layout Factory Methods*
- Example for usage of paperspace viewports: viewports_in_paperspace.py

6.4.14 Blocks

Blocks are collections of DXF entities which can be placed multiple times as block references in different layouts and other block definitions. The block reference (`Insert`) can be rotated, scaled, placed in 3D space by *OCS* and arranged in a grid like manner, each `Insert` entity can have individual attributes (`Attrib`) attached.

Block Attributes

A block attribute (`Attrib`) is a text annotation attached to a block reference with an associated tag. Attributes are often used to add information to block references which can be evaluated and exported by CAD applications.

Extended Block Features

Autodesk added many new features to BLOCKS (dynamic blocks, constraints) as undocumented DXF entities, many of these features are not fully supported by other CAD application and *ezdxf* also has no support or these features beyond the preservation of these undocumented DXF entities.

See also:

Tutorial for Blocks

6.4.15 Layout Extents and Limits

The *extents* and *limits* of an layout represents borders which can be referenced by the ZOOM command or read from some header variables from the `HeaderSection`, if the creator application maintains these values – *ezdxf* does this not automatically.

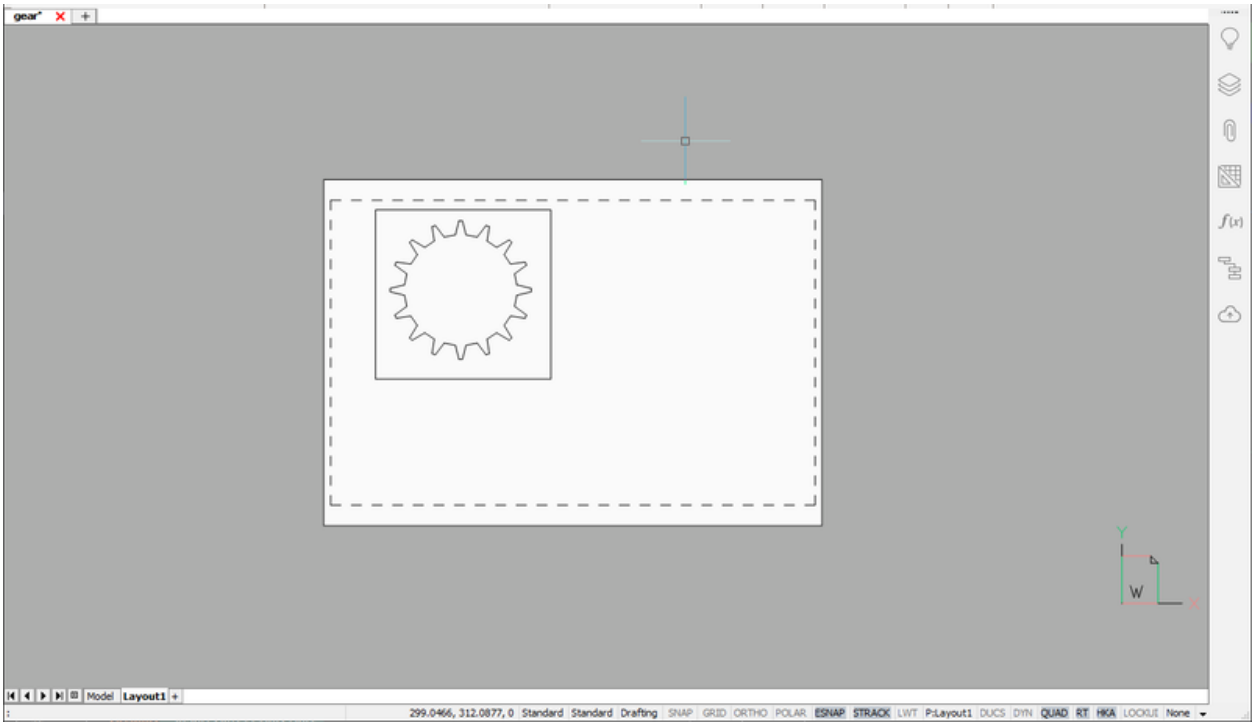
Extents

The *extents* of an layout are determined by the maximum extents of all DXF entities that are in this layout. The command:

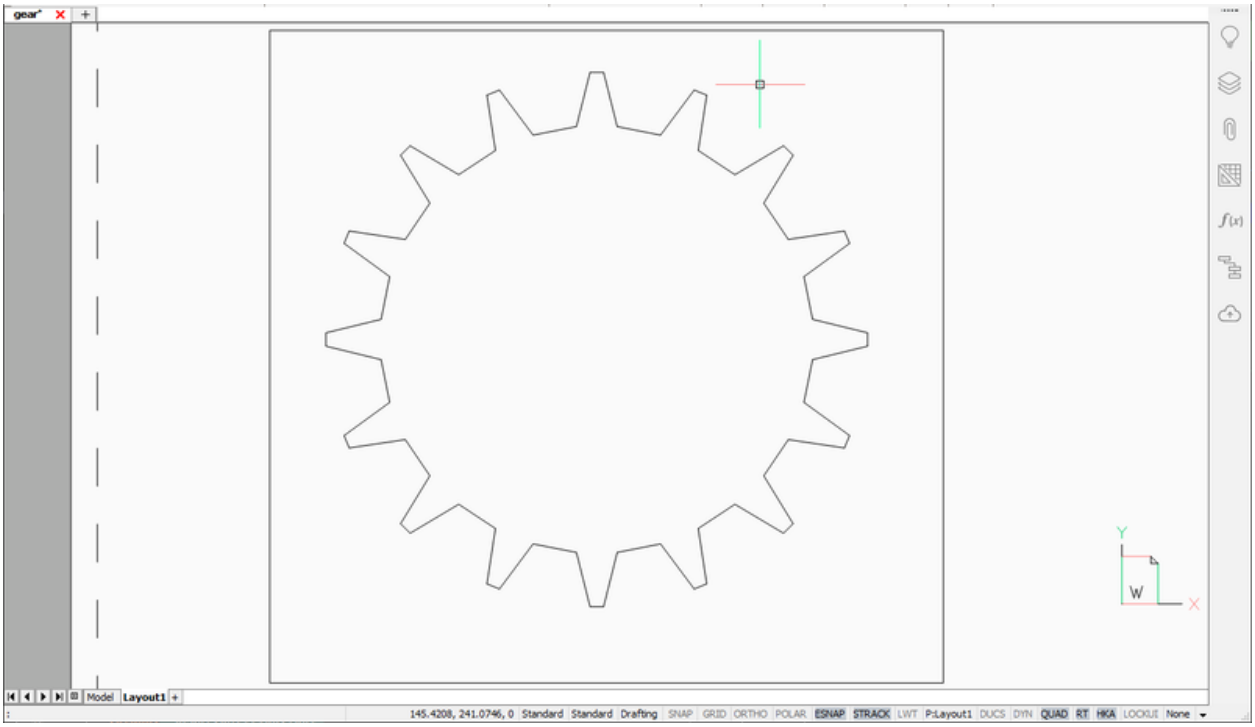
```
ZOOM extents
```

sets the current viewport to the extents of the currently selected layout.

A paperspace layout in an arbitrary zoom state:



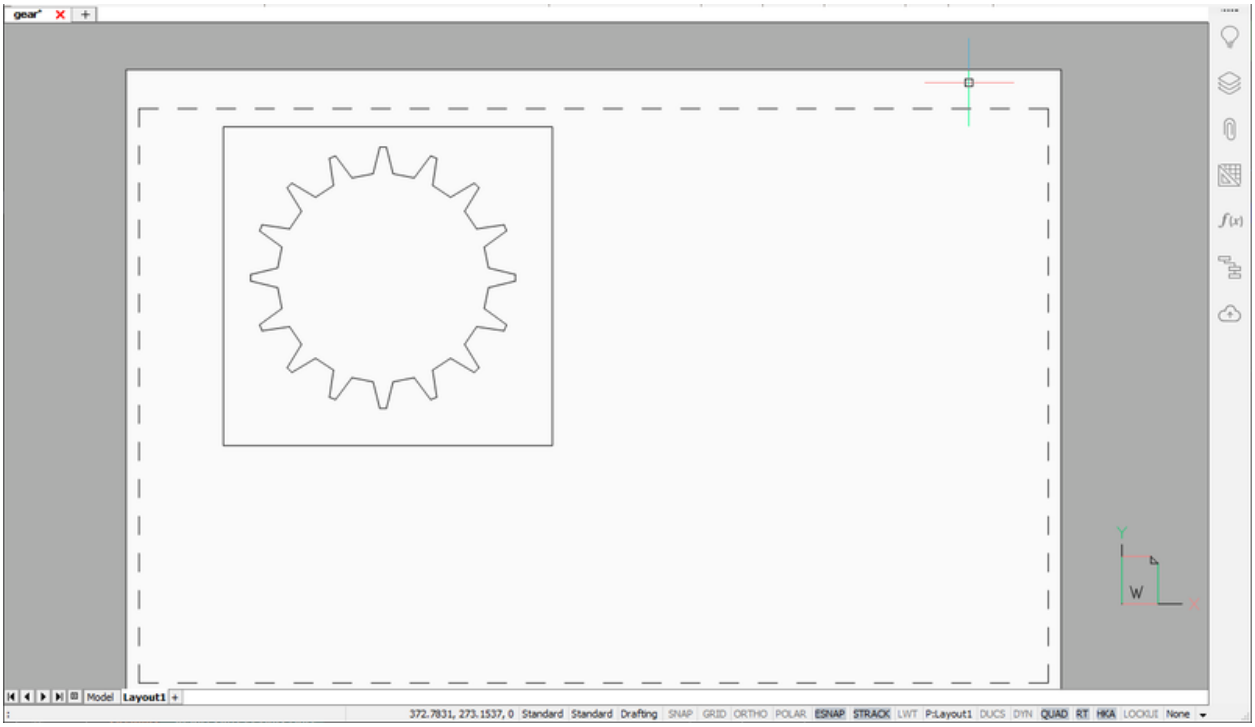
The same layout after the `ZOOM extents` command:



Limits

Sets an invisible rectangular boundary in the drawing area that can limit the grid display and limit clicking or entering point locations. The default limits for paperspace layouts is defined by the paper size.

The layout from above after the `ZOOM all` command:



See also:

The AutoCAD online reference for the ZOOM and the LIMITS command.

Read Stored Values

The extents of the modelspace (the tab called "Model") are stored in the header variable `$EXTMIN` and `$EXTMAX`. The default values of `$EXTMIN` is (+1e20, +1e20, +1e20) and `$EXTMAX` is (-1e20, -1e20, -1e20), which do not describe real borders. These values are copies of the extents attributes of the *Layout* object as `Layout.dxf.extmin` and `Layout.dxf.extmax`.

The limits of the modelspace are stored in the header variables `$LIMMIN` and `$LIMMAX` and have default values of (0, 0) and (420, 297), the default paper size of *ezdxf* in drawing units. These are copies of the *Layout* attributes `Layout.dxf.extmin` and `Layout.dxf.extmax`.

The extents and the limits of the *actual* paperspace layout, which is the last activated paperspace layout tab, are stored in the header variable `$PEXTMIN`, `$PEXTMAX`, `$PLIMMIN` and `$PLIMMAX`.

Each paperspace layout has its own values stored in the *Layout* attributes `Layout.dxf.extmin`, `Layout.dxf.extmax`, `Layout.dxf.limmin` and `Layout.dxf.limmax`.

Setting Extents and Limits

Since v0.16 *ezdxf* it is sufficient to define the attributes for *extents* and *limits* (`Layout.dxf.extmax`, `Layout.dxf.limmin` and `Layout.dxf.limmax`) of *Layout* object. The header variables are synchronized when the document is saved.

The extents of a layout are not calculated automatically by *ezdxf*, as this can take a long time for large documents and correct values are not required to create a valid DXF document.

See also:

How to: *Calculate Extents for the Modelspace*

6.4.16 Font Resources

DXF relies on the infrastructure installed by AutoCAD like the included SHX files or True Type fonts. There is no simple way to store additional information about a used fonts beside the plain file system name like `"arial.ttf"`. The CAD application or viewer which opens the DXF file has to have access to the specified fonts used in your DXF document or it has to use an appropriate replacement font, which is not that easy in the age of unicode. Later DXF versions can store font family names in the XDATA of the STYLE entity but not all CAD application use this information.

6.5 Tutorials

6.5.1 Tutorial for Getting Data from DXF Files

This tutorial shows how to get data from an existing DXF document. If you are a new user of *ezdxf*, read also the tutorial *Usage for Beginners*.

Loading the DXF file:

```
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f"Not a DXF file or a generic I/O error.")
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f"Invalid or corrupted DXF file.")
    sys.exit(2)
```

This works well for DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the *ezdxf.recover* module.

See also:

- *Document Management*
- *Usage for Beginners*

Layouts

The term layout is used as a synonym for an arbitrary entity space which can contain DXF entities like LINE, CIRCLE, TEXT and so on. Each DXF entity can only reside in exact one layout.

There are three different layout types:

- *Modelspace*: the common construction space
- *Paperspace*: used to to create print layouts
- *BlockLayout*: reusable elements, every block has its own entity space

A DXF document consist of exact one modelspace and at least one paperspace. DXF R12 has only one unnamed paperspace the later DXF versions support more than one paperspace and each paperspace has a name.

Getting the modelspace layout

The modelspace contains the "real" world representation of the drawing subjects in real world units. The modelspace has the fixed name "Model" and the DXF document has a special getter method *modelspace()*.

```
msp = doc.modelspace()
```

Iterate over DXF entities of a layout

This code shows how to iterate over all DXF entities in modelspace:

```
# helper function
def print_entity(e):
    print("LINE on layer: %s\n" % e.dxf.layer)
    print("start point: %s\n" % e.dxf.start)
    print("end point: %s\n" % e.dxf.end)

# iterate over all entities in modelspace
msp = doc.modelspace()
for e in msp:
    if e.dxftype() == "LINE":
        print_entity(e)

# entity query for all LINE entities in modelspace
for e in msp.query("LINE"):
    print_entity(e)
```

All layout objects supports the standard Python iterator protocol and the `in` operator.

Access DXF attributes of an entity

The `e.dxftype()` method returns the DXF type, the DXF type is always an uppercase string like `"LINE"`. All DXF attributes of an entity are grouped in the namespace attribute `dxf`:

```
e.dxf.layer  # layer of the entity as string
e.dxf.color  # color of the entity as integer
```

See *Common graphical DXF attributes*

If a DXF attribute is not set (the DXF attribute does not exist), a `DXFValueError` will be raised. The `get()` method returns a default value in this case or `None` if no default value is specified:

```
# If DXF attribute 'paperspace' does not exist, the entity defaults
# to modelspace:
p = e.dxf.get("paperspace", 0)
```

or check beforehand if the attribute exist:

```
if e.dxf.hasattr("paperspace"):
    ...
```

An unsupported DXF attribute raises a `DXFAttributeError`, to check if an attribute is supported by an entity use:

```
if e.dxf.is_supported("paperspace"):
    ...
```

Getting a paperspace layout

```
paperspace = doc.paperspace("layout0")
```

The code above retrieves the paperspace named `layout0`, the usage of the *Paperspace* object is the same as of the modelspace object. DXF R12 provides only one paperspace, therefore the paperspace name in the method call `doc.paperspace("layout0")` is ignored or can be left off. For newer DXF versions you can get a list of the available layout names by the methods *layout_names()* and *layout_names_in_taborder()*.

Retrieve entities by query language

Ezdxf provides a flexible query language for DXF entities. All layout types have a *query()* method to start an entity query or use the *ezdxf.query.new()* function.

The query string is the combination of two queries, first the required entity query and second the optional attribute query, enclosed in square brackets: `"EntityQuery[AttributeQuery]"`

The entity query is a whitespace separated list of DXF entity names or the special name `*`. Where `*` means all DXF entities, all DXF names have to be uppercase. The `*` search can exclude entity types by adding the entity name with a preceding `!` (e.g. `* !LINE`, search all entities except lines).

The attribute query is used to select DXF entities by its DXF attributes. The attribute query is an addition to the entity query and matches only if the entity already match the entity query. The attribute query is a boolean expression, supported operators: `and`, `or`, `!`.

See also:

Entity Query String

Get all LINE entities from the modelspace:

```
msp = doc.modelspace()
lines = msp.query("LINE")
```

The result container *EntityQuery* also provides the `query()` method to further refine the query, such as retrieving all LINE entities at layer `construction`:

```
construction_lines = lines.query('*[layer=="construction"]')
```

The `*` is a wildcard for all DXF types, in this case you could also use `LINE` instead of `*`, `*` works here because the source just contains LINE entities.

This could be executed as a single query:

```
lines = msp.query('LINE[layer=="construction"]')
```

An advanced query for getting all modelspace entities at layer `construction`, but excluding entities with linetype `DASHED`:

```
not_dashed_entities = msp.query('*[layer=="construction" and linetype!="DASHED"]')
```

Extended EntityQuery Features

The *EntityQuery* class has properties and overloaded operators to build extended queries by Python features instead of a query string.

Same task as in the previous section but using features of the *EntityQuery* container:

```
# The overloaded rational operators return an EntityQuery object and not a bool value!
lines = msp.query("LINES").layer == "construction"
not_dashed_lines = lines.linetype != "DASHED"
```

See also:

Extended EntityQuery Features

Retrieve entities by groupby() function

The *groupby()* function searches and group entities by a user defined criteria. As an example let's group all entities from modelspace by layer, the result will be a *dict* with layer names as dict-key and a list of all entities from the modelspace matching this layer as dict-value:

```
from ezdxf.groupby import groupby
group = groupby(entities=msp, dxfattrib="layer")
```

The *entities* argument can be any container or generator which yields DXF entities:

```
group = msp.groupby(dxfattrib="layer")

for layer, entities in group.items():
    print(f'Layer "{layer}" contains following entities:')
    for entity in entities:
        print(f"    {entity}")
    print("-"*40)
```

The previous example shows how to group entities by a single DXF attribute. For a more advanced query create a custom key function, which accepts a DXF entity as argument and returns a hashable value as dict-key or `None` to exclude the entity.

The following example shows how to group entities by layer and color, the dict-key is a `(layer, color)` tuple and the dict-value is a list of entities with matching DXF attributes:

```
def layer_and_color_key(entity):
    # return None to exclude entities from the result container
    if entity.dxf.layer == "0":  # exclude entities from default layer "0"
        return None
    else:
        return entity.dxf.layer, entity.dxf.color

group = msp.groupby(key=layer_and_color_key)
for key, entities in group.items():
    print(f'Grouping criteria "{key}" matches following entities:')
    for entity in entities:
        print(f"    {entity}")
    print("-"*40)
```

The *groupby()* function catches `DXFAttributeError` exceptions while processing entities and excludes this entities from the result. There is no need to worry about DXF entities which do not support certain attributes, they will be excluded automatically.

See also:

groupby() documentation

6.5.2 Tutorial for Creating DXF Drawings

Create a new DXF document by the *ezdxf.new()* function:

```
import ezdxf

# create a new DXF R2010 document
doc = ezdxf.new("R2010")

# add new entities to the modelspace
msp = doc.modelspace()
# add a LINE entity
msp.add_line((0, 0), (10, 0))
# save the DXF document
doc.saveas("line.dxf")
```

New entities are always added to layouts, a layout can be the modelspace, a paperspace layout or a block layout.

See also:

Thematic Index of Layout Factory Methods

Predefined Resources

Ezdxf creates new DXF documents with as little content as possible, this means only the resources that are absolutely necessary are created. The *ezdxf.new()* function can create some standard resources, such as linetypes and text styles, by setting the argument *setup* to `True`.

```
import ezdxf

doc = ezdxf.new("R2010", setup=True)
msp = doc.modelspace()
msp.add_line((0, 0), (10, 0), dxfattribs={"linetype": "DASHED"})
```

The defined standard linetypes are shown in the basic concept section for *Linetypes* and the available text styles are shown in the *Tutorial for Text*.

Important: To see the defined text styles in a DXF viewer or CAD application, the applications have to know where the referenced TTF fonts can be found. This configuration is not possible by *ezdxf* and has to be done for each application as described in their documentation.

See also: *Font Resources*

Simple DXF R12 drawings

The *r12writer* add-on creates simple DXF R12 drawings with a restricted set of DXF types: LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE.

The advantage of the *r12writer* is the speed and the small memory footprint, all entities are written directly to a file or stream without creating a document structure in memory.

See also:

r12writer

6.5.3 Tutorial for Common Graphical Attributes

The graphical attributes `color`, `linetype`, `lineweight`, `true_color`, `transparency`, `ltscale` and `invisible` are available for all graphical DXF entities and are located in the DXF namespace attribute `dxf` of the DXF entities. All these attributes are optional and all except for `true_color` and `transparency` have a default value.

Not all of these attributes are supported by all DXF versions. This table shows the minimum required DXF version for each attribute:

R12	`color`, `linetype`
R2000	`lineweight`, `ltscale`, `invisible`
R2004	`true_color`, `transparency`

Color

Please read the section about the *AutoCAD Color Index (ACI)* to understand the basics.

The usage of the *color* attribute is very straight forward. Setting the value is:

```
entity.dxf.color = 1
```

and getting the value looks like this:

```
value = entity.dxf.color
```

The `color` attribute has a default value of 256, which means take the color defined by the layer associated to the entity. The *ezdxf.colors* module defines some constants for often used color values:

```
entity.dxf.color = ezdxf.colors.RED
```

The *ezdxf.colors.aci2rgb()* function converts the ACI value to the RGB value of the default modelspace palette.

See also:

- Basics about *AutoCAD Color Index (ACI)*
- *ezdxf.colors* module

True Color

Please read the section about *True Color* to understand the basics.

The easiest way is to use the `rgb` property to set and get the true color values as RGB tuples:

```
entity.rgb = (255, 128, 16)
```

The `rgb` property return `None` if the `true_color` attribute is not present:

```
rgb = entity.rgb
if rgb is not None:
    r, g, b = rgb
```

Setting and getting the `true_color` DXF attribute directly is possible and the *ezdxf.colors* module has helper function to convert RGB tuples to 24-bit value and back:

```
entity.dxf.true_color = ezdxf.colors.rgb2int(255, 128, 16)
```

The `true_color` attribute is optional does not have a default value and therefore it is not safe to use the attribute directly, check if the attribute exists beforehand:

```
if entity.dxf.hasattr("true_color"):
    r, g, b = ezdxf.colors.int2rgb(entity.dxf.true_color)
```

or use the `get()` method of the `dxf` namespace attribute to get a default value if the attribute does not exist:

```
r, g, b = ezdxf.colors.int2rgb(entity.dxf.get("true_color", 0)
```

See also:

- Basics about *True Color*
- *ezdxf.colors* module

Transparency

Please read the section about *Transparency* to understand the basics.

It's recommended to use the *transparency* property of the *DXFGraphic* base class. The `transparency` property is a float value in the range from 0.0 to 1.0 where 0.0 is opaque and 1.0 if fully transparent:

```
entity.transparency = 0.5
```

or set the values of the DXF attribute by constants defined in the *ezdxf.colors* module:

```
entity.dxf.transparency = ezdxf.colors.TRANSPARENCY_50
```

The default setting for `transparency` in CAD applications is always transparency by layer, but the `transparency` property in *ezdxf* has a default value of 0.0 (opaque), so there are additional entity properties to check if the transparency value should be taken from the associated entity layer or from the parent block:

```
if entity.is_transparency_by_layer:
    ...
elif entity.is_transparency_by_block:
    ...
else:
    ...
```

The top level entity attribute `transparency` does not support setting transparency by layer or block:

```
from ezdxf import colors

...

# set transparency by layer by removing the DXF attribute "transparency":
entity.dxf.discard("transparency")

# set transparency by block:
entity.dxf.transparency = colors.TRANSPARENCY_BYBLOCK

# there are also some handy constants in the colors module:
# TRANSPARENCY_10 upto TRANSPARENCY_90 in steps of 10
entity.dxf.transparency = colors.TRANSPARENCY_30  # set 30% transparency
entity.dxf.transparency = colors.OPAQUE
```

See also:

- Basics about *Transparency*
- *ezdxf.colors* module

Linetype

Please read the section about *Linetypes* to understand the basics.

The `linetype` attribute contains the name of the linetype as string and can be set by the `dxf` namespace attribute directly:

```
entity.dxf.linetype = "DASHED"  # linetype DASHED must exist!
```

The `linetype` attribute is optional and has a default value of "BYLAYER", so the attribute can always be used without any concerns:

```
name = entity.dxf.linetype
```

Warning: Make sure the linetype you assign to an entity is really defined in the linetype table otherwise AutoCAD will not open the DXF file. There are no implicit checks for that by *ezdxf* but you can call the *audit()* method of the DXF document explicitly to validate the document before exporting.

Ezdxf creates new DXF documents with as little content as possible, this means only the resources that are absolutely necessary are created. The *ezdxf.new()* function can create some standard linetypes by setting the argument *setup* to `True`:

```
doc = ezdxf.new("R2010", setup=True)
```

See also:

- Basics about *Linetypes*
- *Tutorial for Creating Linetype Pattern*

Lineweight

Please read the section about *Lineweights* to understand the basics.

The `lineweight` attribute contains the lineweight as an integer value and can be set by the `dxf` namespace attribute directly:

```
entity.dxf.lineweight = 25
```

The `lineweight` value is the line width in millimeters times 100 e.g. 0.25mm = 25, but only certain values are valid for more information go to section: *Lineweights*.

Values < 0 have a special meaning and can be imported as constants from *ezdxf.lldxf.const*

-1	LINEWEIGHT_BYLAYER
-2	LINEWEIGHT_BYBLOCK
-3	LINEWEIGHT_DEFAULT

The `lineweight` attribute is optional and has a default value of -1, so the attribute can always be used without any concerns:

```
lineweight = entity.dxf.lineweight
```

Important: You have to enable the option to show lineweights in your CAD application or viewer to see the effect on screen, which is disabled by default, the same has to be done in the page setup options for plotting lineweights.

```
# activate on screen lineweight display
doc.header["$LWDISPLAY"] = 1
```

See also:

- Basics about *Lineweights*

Linetype Scale

The `ltscale` attribute scales the linetype pattern by a float value and can be set by the `dxf` namespace attribute directly:

```
entity.dxf.ltscale = 2.0
```

The `ltscale` attribute is optional and has a default value of 1.0, so the attribute can always be used without any concerns:

```
scale = entity.dxf.ltscale
```

See also:

- Basics about *Linetypes*

Invisible

The `invisible` attribute an boolean value (0/1) which defines if an entity is invisible or visible and can be set by the `dxf` namespace attribute directly:

```
entity.dxf.invisible = 1
```

The `invisible` attribute is optional and has a default value of 0, so the attribute can always be used without any concerns:

```
is_invisible = bool(entity.dxf.invisible)
```

GfxAttribs

When adding new entities to an entity space like the modelspace or a block definition, the factory methods expect the graphical DXF attributes by the argument *dxfattribs*. This object can be a Python `dict` where the key is the DXF attribute name and the value is the attribute value, or better use the *GfxAttribs* object which has some additional validation checks and support for code completions by IDEs:

```
import ezdxf
from ezdxf.gfxattribs import GfxAttribs

doc = ezdxf.new()
msp = doc.modelspace()

line = msp.add_line(
    (0, 0), (10, 10), dxfattribs=GfxAttribs(layer="0", rgb=(25, 128, 16))
)
```

See also:

- *ezdxf.gfxattribs* module

6.5.4 Tutorial for Layers

If you are not familiar with the concept of layers, please read this first: Concept of *Layers*

Reminder: a layer definition is not required for using a layer!

Create a Layer Definition

```
import ezdxf

doc = ezdxf.new(setup=True)  # setup required line types
msp = doc.modelspace()
doc.layers.add(name="MyLines", color=7, linetype="DASHED")
```

The advantage of assigning a linetype and a color to a layer is that entities on this layer can inherit this properties by using `"BYLAYER"` as linetype string and `256` as color, both values are default values for new entities so you can leave off these assignments:

```
msp.add_line((0, 0), (10, 0), dxfattribs={"layer": "MyLines"})
```

The new created line will be drawn with color `7` and linetype `"DASHED"`.

Moving an Entity to a Different Layer

Moving an entity to a different layer is a simple assignment of the new layer name to the `layer` attribute of the entity.

```
line = msp.add_line((0, 0), (10, 0), dxfattribs={"layer": "MyLines"})
# move the entity to layer "OtherLayer"
line.dxf.layer = "OtherLayer"
```

Changing Layer State

Get the layer definition object from the layer table:

```
my_lines = doc.layers.get('MyLines')
```

Check the state of the layer:

```
my_lines.is_off()  # True if layer is off
my_lines.is_on()   # True if layer is on
my_lines.is_locked()  # True if layer is locked
layer_name = my_lines.dxf.name  # get the layer name
```

Change the state of the layer:

```
# switch layer off, entities at this layer will not shown in CAD applications/viewers
my_lines.off()

# lock layer, entities at this layer are not editable in CAD applications
my_lines.lock()
```

Get/set the color of a layer by property `Layer.color`, because the DXF attribute `Layer.dxf.color` is misused for switching the layer on and off, the layer is off if the color value is negative.

Changing the layer properties:

```
my_lines.dxf.linetype = "DOTTED"
my_lines.color = 13  # preserves on/off state of layer
```

See also:

For all methods and attributes see class *Layer*.

Check Available Layers

The *LayerTable* object supports some standard Python protocols:

```
# iteration
for layer in doc.layers:
    if layer.dxf.name != "0":
        layer.off()  # switch all layers off except layer "0"

# check for existing layer definition
if "MyLines" in doc.layers:
    layer = doc.layers.get("MyLines")

layer_count = len(doc.layers) # total count of layer definitions
```

Renaming a Layer

The *Layer* class has a method for renaming the layer, but has same limitations, not all places where layer references can occur are documented, third-party entities are black-boxes with unknown content and layer references could be stored in the extended data section of any DXF entity or in a XRECORD entity, so some references may reference a non-existing layer definition after the renaming, at least these references are still valid, because a layer definition is not required for using a layer.

```
my_lines = doc.layers.get("MyLines")
my_lines.rename("YourLines")
```

Deleting a Layer Definition

Delete a layer definition:

```
doc.layers.remove("MyLines")
```

This just deletes the layer definition, all DXF entities referencing this layer still exist, if they inherit any properties from the deleted layer they will now get the default layer properties.

Warning: The behavior of entities referencing the layer by handle is unknown and may break the DXF document.

Deleting All Entities From a Layer

Because of all these uncertainties about layer references mentioned above, deleting all entities referencing a certain layer from a DXF document is not implemented as an API call!

Nonetheless deleting all graphical entities from the DXF document which do reference a certain layer by the `layer` attribute is a safe procedure:

```
key_func = doc.layers.key
layer_key = key_func("MyLines")
# The trashcan context-manager is a safe way to delete entities from the
# entities database while iterating.
with doc.entitydb.trashcan() as trash:
    for entity in doc.entitydb.values():
        if not entity.dxf.hasattr("layer"):
            continue
        if layer_key == key_func(entity.dxf.layer):
            # safe destruction while iterating
            trash.add(entity.dxf.handle)
```

6.5.5 Tutorial for Creating Linetype Pattern

Simple line type example:

You can define your own linetypes. A linetype definition has a name, a description and line pattern elements:

```
elements = [total_pattern_length, elem1, elem2, ...]
```

total_pattern_length
Sum of all linetype elements (absolute values)

elem
if elem > 0 it is a line, if elem < 0 it is gap, if elem == 0.0 it is a dot

Create a new linetype definition:

```
import ezdxf
from ezdxf.tools.standards import linetypes  # some predefined linetypes

doc = ezdxf.new()
msp = doc.modelspace()

my_line_types = [
    (
        "DOTTED",
        "Dotted .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .",
        [0.2, 0.0, -0.2],
    ),
    (
        "DOTTEDX2",
        "Dotted (2x) .    .    .    .    .    .    .    . ",
        [0.4, 0.0, -0.4],
    ),
    (
        "DOTTED2",
        "Dotted (.5) . . . . . . . . . . . . . . . . . . . . ",
        [0.1, 0.0, -0.1],
    ),
]
for name, desc, pattern in my_line_types:
    if name not in doc.linetypes:
        doc.linetypes.add(
            name=name,
            pattern=pattern,
            description=desc,
        )
```

Setup some predefined linetypes:

```
for name, desc, pattern in linetypes():
    if name not in doc.linetypes:
        doc.linetypes.add(
            name=name,
            pattern= pattern,
            description=desc,
        )
```

Check Available Linetypes

The linetypes object supports some standard Python protocols:

```
# iteration
print("available linetypes:")
for lt in doc.linetypes:
    print(f"{lt.dxf.name}: {lt.dxf.description}")

# check for existing linetype
if "DOTTED" in doc.linetypes:
    pass

count = len(doc.linetypes) # total count of linetypes
```

Removing Linetypes

Warning: Ezdxf does not check if a linetype is still in use and deleting a linetype which is still in use generates an **invalid** DXF file. The audit process `audit()` of the DXF document removes `linetype` attributes referencing non existing linetypes.

You can delete a linetype:

```
doc.layers.remove("DASHED")
```

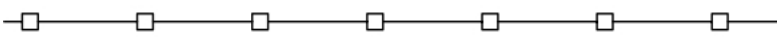
This just removes the linetype definition, the `linetype` attribute of DXF entities may still refer the removed linetype definition "DASHED" and AutoCAD will not open DXF files including undefined linetypes.

6.5.6 Tutorial for Creating Complex Linetype Pattern

In DXF R13 Autodesk introduced complex linetypes, containing TEXT or SHAPES in line types.

Complex linetype example with text:

Complex line type example with shapes:



For easy usage the pattern string for complex line types is mostly the same string as the pattern definition strings in AutoCAD ".lin" files.

Example for complex line type TEXT:

```
doc = ezdxf.new("R2018")  # DXF R13 or later is required

doc.linetypes.add(
    name="GASLEITUNG2",
    # linetype definition string from acad.lin:
    pattern='A,.5,-.2,["GAS",STANDARD,S=.1,U=0.0,X=-0.1,Y=-.05],-.25',
    description= "Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----",
```

(continues on next page)

(continued from previous page)

```
    length=1,  # required for complex line types
})
```

The pattern always starts with an "A", the following float values have the same meaning as for simple linetypes, a value > 0 is a line, a value < 0 is a gap, and a 0 is a point, the opening square bracket "[" starts the complex part of the linetype pattern.

The text after the "[" defines the complex linetype:

- A text in quotes (e.g. "GAS") defines a *complex TEXT linetype* and represents the pattern text itself.
- A text without quotes is a SHAPE name (in ".lin" files) and defines a *complex SHAPE linetype. Ezdxf can not translate this SHAPE name from the ".lin" file into the required shape file index, so *YOU* have to translate this SHAPE name into the shape file index, e.g. saving the file with AutoCAD as DXF and searching for the DXF linetype definition, see example below and the DXF Internals: *LTYPE Table*.

For *complex TEXT linetypes* the second parameter is the text style, for *complex SHAPE linetypes* the second parameter is the shape file name, the shape file has to be in the same directory as the DXF file or in one of the CAD application support paths.

The meaning of the following comple linetype parameters are shown in the table below:

S	scaling factor, always > 0, if S=0 the TEXT or SHAPE is not visible
R or U	rotation relative to the line direction
X	x-direction offset (along the line)
Y	y-direction offset (perpendicular to the line)

These parameters are case insensitive and the closing square bracket "]" ends the complex part of the linetype pattern.

The fine tuning of this parameters is a try an error process, for *complex TEXT linetypes* the scaling factor (e.g. the STANDARD text style) sets the text height (e.g. "S=0.1" sets the text height to 0.1 units), by shifting in y-direction by half of the scaling factor, the text is vertically centered to the line. For the x-direction it seems to be a good practice to place a gap in front of the text and after the text, find x shifting value and gap sizes by try and error. The overall length is at least the sum of all line and gap definitions (absolute values).

Example for complex line type SHAPE:

```
doc.linetypes.add("GRENZE2",
    # linetype definition in acad.lin:
    # A,.25,-.1,[BOX,ltypeshp.shx,x=-.1,s=.1],-.1,1
    # replacing BOX by shape index 132 (got index from an AutoCAD file),
    # ezdxf can't get shape index from ltypeshp.shx
    pattern="A,.25,-.1,[132,ltypeshp.shx,x=-.1,s=.1],-.1,1",
    description="Grenze eckig ----[]-----[]----[]-----[]----[]--",
    length= 1.45,  # required for complex line types
})
```

Complex line types with shapes only work if the associated shape file (e. g. ltypeshp.shx) and the DXF file are in the same directory or the shape file is placed in one of the CAD application support folders.

6.5.7 Tutorial for Simple DXF Entities

These are basic graphical entities located in an entity space like the modelspace or a block definition and only support the common graphical attributes.

The entities in the following examples are always placed in the xy-plane of the *WCS* aka the 2D drawing space. Some of these entities can only be placed outside the xy-plane in 3D space by utilizing the *OCS*, but this feature is beyond the scope of this tutorial, for more information about that go to: *Tutorial for OCS/UCS Usage*.

Prelude to all following examples:

```
import ezdxf
from ezdxf.gfxattribs import GfxAttribs

doc = ezdxf.new()
doc.layers.new("ENTITY", color=1)
msp = doc.modelspace()
attribs = GfxAttribs(layer="ENTITY")
```

See also:

- *Tutorial for Creating DXF Drawings*
- *Tutorial for Layers*
- `ezdxf.gfxattribs` module

Point

The `Point` entity marks a 3D point in the *WCS*:

```
point = msp.add_point((10, 10), dxfattribs=attribs)
```

All `Point` entities have the same styling stored in the header variable $PDMODE, for more information read the reference of class `Point`.

See also:

- Reference of class `Point`
- *Tutorial for Common Graphical Attributes*

Line

The `Line` entity is a 3D line with a start- and an end point in the *WCS*:

```
line = msp.add_line((0, 0), (10, 10), dxfattribs=attribs)
```

See also:

- Reference of class `Line`
- *Tutorial for Common Graphical Attributes*
- `ezdxf.math.ConstructionLine`

Circle

The *Circle* entity is an *OCS* entity defined by a center point and a radius:

```
circle = msp.add_circle((10, 10), radius=3, dxfattribs=attribs)
```

See also:

- Reference of class *Circle*
- *Tutorial for Common Graphical Attributes*
- *ezdxf.math.ConstructionCircle*

Arc

The *Arc* entity is an *OCS* entity defined by a center point, a radius a start- and an end angle in degrees:

```
arc = msp.add_arc((10, 10), radius=3, start_angle=30, end_angle=120,␣
↪dxfattribs=attribs)
```

The arc goes always in counter-clockwise orientation around the z-axis more precisely the extrusion vector of *OCS*, but this is beyond the scope of this tutorial.

The helper class *ezdxf.math.ConstructionArc* provides constructors to create arcs from different scenarios:

- *from_2p_angle*: arc from 2 points and an angle
- *from_2p_radius*: arc from 2 points and a radius
- *from_3p*: arc from 3 points

This example creates an arc from point (10, 0) to point (0, 0) passing the point (5, 3):

```
from ezdxf.math import ConstructionArc

# -x-x-x- snip -x-x-x-

arc = ConstructionArc.from_3p(
    start_point=(10, 0), end_point=(0, 0), def_point=(5, 3)
)
arc.add_to_layout(msp, dxfattribs=attribs)
```

See also:

- Reference of class *Arc*
- *Tutorial for Common Graphical Attributes*
- *ezdxf.math.ConstructionArc*

Ellipse

The *Ellipse* entity requires DXF R2000 or newer and is a true *WCS* entity. The ellipse is defined by a center point, a vector for the major axis, the ratio between major- and minor axis and the start- and end parameter in radians:

```
ellipse = msp.add_ellipse(
    (10, 10), major_axis=(5, 0), ratio=0.5, start_param=0, end_param=math.pi,␣
↪dxfattribs=attribs
)
```

When placed in 3D space the extrusion vector defines the normal vector of the ellipse plane and the minor axis is the extrusion vector `cross` the major axis.

See also:

- Reference of class *Ellipse*
- *Tutorial for Common Graphical Attributes*
- *ezdxf.math.ConstructionEllipse*

Further Tutorials

- *Tutorial for LWPolyline*
- *Tutorial for Spline*
- *Tutorial for Text*
- *Tutorial for MText and MTextEditor*
- *Tutorial for Hatch*
- *Tutorial for MultiLeader*
- *Tutorial for Mesh*

6.5.8 Tutorial for Blocks

If you are not familiar with the concept of blocks, please read this first: Concept of *Blocks*

Create a Block

Blocks are managed as *BlockLayout* objects by the *BlocksSection* object, every drawing has only one blocks section referenced by attribute `Drawing.blocks`.

```
import ezdxf
import random  # needed for random placing points


def get_random_point():
    """Returns random x, y coordinates."""
    x = random.randint(-100, 100)
    y = random.randint(-100, 100)
    return x, y
```

(continues on next page)

(continued from previous page)

```
# Create a new drawing in the DXF format of AutoCAD 2010
doc = ezdxf.new('R2010')

# Create a block with the name 'FLAG'
flag = doc.blocks.new(name='FLAG')

# Add DXF entities to the block 'FLAG'.
# The default base point (= insertion point) of the block is (0, 0).
flag.add_lwpolyline([(0, 0), (0, 5), (4, 3), (0, 3)])  # the flag symbol as 2D␣
↪polyline
flag.add_circle((0, 0), .4, dxfattribs={'color': 2})  # mark the base point with a␣
↪circle
```

Block References (Insert)

A block reference can be created by adding an *Insert* entity to any of these layout types:

- *Modelspace*
- *Paperspace*
- *BlockLayout*

A block reference can be scaled and rotated individually. Lets add some random flags to the modelspace:

```
# Get the modelspace of the drawing.
msp = doc.modelspace()

# Get 50 random placing points.
placing_points = [get_random_point() for _ in range(50)]

for point in placing_points:
    # Every flag has a different scaling and a rotation of -15 deg.
    random_scale = 0.5 + random.random() * 2.0
    # Add a block reference to the block named 'FLAG' at the coordinates 'point'.
    msp.add_blockref('FLAG', point, dxfattribs={
        'xscale': random_scale,
        'yscale': random_scale,
        'rotation': -15
    })

# Save the drawing.
doc.saveas("blockref_tutorial.dxf")
```

Query all block references of block `FLAG`:

```
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    print(str(flag_ref))
```

When adding a block reference to a layout with different units, the scaling factor between these units should be applied as scaling attributes (`xscale`, ...) e.g. modelspace in meters and block in centimeters, `xscale` has to be 0.01.

Block Attributes

A block attribute (*Attrib*) is a text annotation attached to a block reference with an associated tag. Attributes are often used to add information to blocks which can be evaluated and exported by CAD applications. An attribute can be added to a block reference by the `Insert.add_attrib()` method, the ATTRIB entity is geometrically not related to the block reference, so insertion point, rotation and scaling of the attribute have to be calculated by the user, but helper tools for that do exist.

Using Attribute Definitions

Another way to add attributes to block references is using attribute templates (*AttDef*). First create the attribute definition in the block definition, then add the block reference by `add_blockref()` and attach and fill attributes automatically by the *add_auto_attribs()* method to the block reference. This method has the advantage that all attributes are placed relative to the block base point with the same rotation and scaling as the block reference, but non-uniform scaling is not handled very well.

The *add_auto_blockref()* method handles non-uniform scaling better by wrapping the block reference and its attributes into an anonymous block and let the CAD application do the transformation work. This method has the disadvantage of a more complex evaluation of attached attributes

Using attribute definitions (*AttDef* templates):

```
# Define some attributes for the block 'FLAG', placed relative
# to the base point, (0, 0) in this case.
flag.add_attdef('NAME', (0.5, -0.5), dxfattribs={'height': 0.5, 'color': 3})
flag.add_attdef('XPOS', (0.5, -1.0), dxfattribs={'height': 0.25, 'color': 4})
flag.add_attdef('YPOS', (0.5, -1.5), dxfattribs={'height': 0.25, 'color': 4})

# Get another 50 random placing points.
placing_points = [get_random_point() for _ in range(50)]

for number, point in enumerate(placing_points):
    # values is a dict with the attribute tag as item-key and
    # the attribute text content as item-value.
    values = {
        'NAME': "P(%d)" % (number + 1),
        'XPOS': "x = %.3f" % point[0],
        'YPOS': "y = %.3f" % point[1]
    }

    # Every flag has a different scaling and a rotation of +15 deg.
    random_scale = 0.5 + random.random() * 2.0
    blockref = msp.add_blockref('FLAG', point, dxfattribs={
        'rotation': 15
    }).set_scale(random_scale)
    blockref.add_auto_attribs(values)

# Save the drawing.
doc.saveas("auto_blockref_tutorial.dxf")
```

Get/Set Attributes of Existing Block References

See the howto: *Get/Set Block Reference Attributes*

Evaluate Wrapped Block References

As mentioned above the evaluation of block references wrapped into anonymous blocks is complex:

```
# Collect all anonymous block references starting with '*U'
anonymous_block_refs = modelspace.query('INSERT[name ? "^\*U.+"]')

# Collect the references of the 'FLAG' block
flag_refs = []
for block_ref in anonymous_block_refs:
    # Get the block layout of the anonymous block
    block = doc.blocks.get(block_ref.dxf.name)
    # Find all block references to 'FLAG' in the anonymous block
    flag_refs.extend(block.query('INSERT[name=="FLAG"]'))

# Evaluation example: collect all flag names.
flag_numbers = [
    flag.get_attrib_text("NAME")
    for flag in flag_refs
    if flag.has_attrib("NAME")
]

print(flag_numbers)
```

Exploding Block References

This is an advanced feature and the results may not be perfect. A **non-uniform scaling** lead to incorrect results for text entities (TEXT, MTEXT, ATTRIB) and some other entities like HATCH with circular- or elliptic path segments. The "exploded" entities are added to the same layout as the block reference by default.

```
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    flag_ref.explode()
```

Examine Entities of Block References

To just examine the content entities of a block reference use the *virtual_entities()* method. This methods yields "virtual" entities with properties identical to "exploded" entities but they are not stored in the entity database, have no handle and are not assigned to any layout.

```
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    for entity in flag_ref.virtual_entities():
        if entity.dxftype() == "LWPOLYLINE":
            print(f"Found {str(entity)}.")
```

6.5.9 Tutorial for LWPolyline

The *LWPolyline* (lightweight polyline) was introduced in DXF R13/14 and it is defined as a single graphic entity, which differs from the old-style *Polyline* entity, which is defined as a group of sub-entities. It is recommended to prefer the LWPOLYLINE over the 2D POLYLINE entity because it requires less space in memory and in DXF files and displays faster in AutoCAD.

Important: The LWPOLYLINE is a planar element, therefore the (x, y) point coordinates are located in the *OCS* and the z-axis is stored in the `LWPolyline.dxf.elevation` attribute. The method *vertices_in_wcs* returns the polyline vertices as WCS coordinates.

Create a simple polyline:

```
import ezdxf

doc = ezdxf.new("R2000")
msp = doc.modelspace()

points = [(0, 0), (3, 0), (6, 3), (6, 6)]
msp.add_lwpolyline(points)

doc.saveas("lwpolyline1.dxf")
```

Append multiple points to a polyline:

```
doc = ezdxf.readfile("lwpolyline1.dxf")
msp = doc.modelspace()

line = msp.query("LWPOLYLINE").first
if line is not None:
    line.append_points([(8, 7), (10, 7)])

doc.saveas("lwpolyline2.dxf")
```

The index operator `[]` always returns polyline points as 5-tuple (x, y, start_width, end_width, bulge), the start_width, end_width and bulge values are 0 if not present:

```
first_point = line[0]
x, y, start_width, end_width, bulge = first_point
```

The context manager *points()* can be used to edit polyline points, this method was introduced because accessing individual points was very slow in early versions of *ezdxf*, in current versions of *ezdxf* the direct access by the index operator `[]` is very fast and using the context manager is not required anymore, but the context manager still exist and has the advantage of supporting an user defined point format:

```
doc = ezdxf.readfile("lwpolyline2.dxf")
msp = doc.modelspace()

line = msp.query("LWPOLYLINE").first

with line.points("xyseb") as points:
    # points is a standard Python list
    # existing points are 5-tuples, but new points can be
    # set as (x, y, [start_width, [end_width, [bulge]]]) tuple
    # set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).
```

(continues on next page)

(continued from previous page)

```
    # delete last 2 points
    del points[-2:]
    # adding two points
    points.extend([(4, 7), (0, 7)])

doc.saveas("lwpolyline3.dxf")
```

Each line segment can have a different start- and end width, if omitted start- and end width is 0:

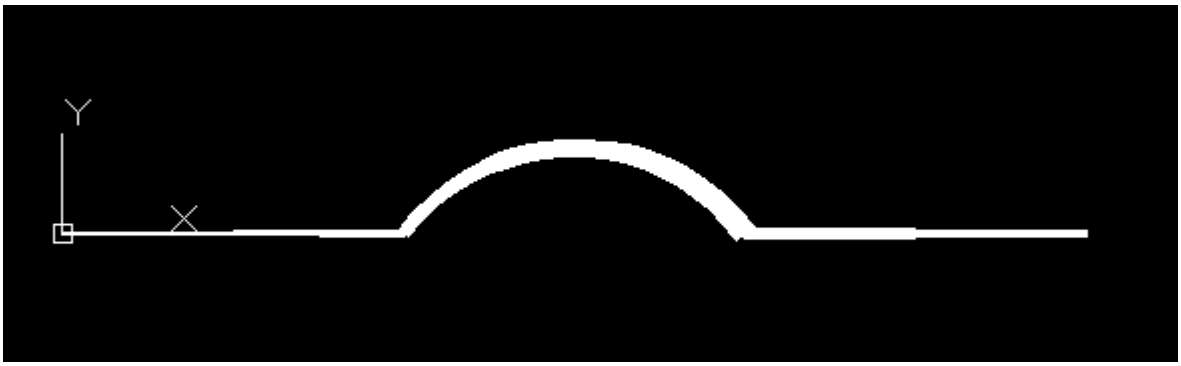
```
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# point format = (x, y, [start_width, [end_width, [bulge]]])
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

points = [(0, 0, .1, .15), (3, 0, .2, .25), (6, 3, .3, .35), (6, 6)]
msp.add_lwpolyline(points)

doc.saveas("lwpolyline4.dxf")
```

The first point carries the start- and end-width of the first segment, the second point of the second segment and so on, the start- and end width value of the last point is used for the closing segment if the polyline is closed else these values are ignored. Start- and end width only works if the DXF attribute `dxf.const_width` is unset, delete it to be sure it's unset:

```
# no exception will be raised if const_width is already unset:
del line.dxf.const_width
```

`LWPolyline` can also have curved elements, they are defined by the *Bulge value*:

```
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# point format = (x, y, [start_width, [end_width, [bulge]]])
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

points = [(0, 0, 0, .05), (3, 0, .1, .2, -.5), (6, 0, .1, .05), (9, 0)]
msp.add_lwpolyline(points)

doc.saveas("lwpolyline5.dxf")
```



The curved segment is drawn from the point which defines the *bulge* value to the following point, the curved segment is always an arc. The bulge value defines the ratio of the arc sagitta (segment height *h*) to half line segment length (point

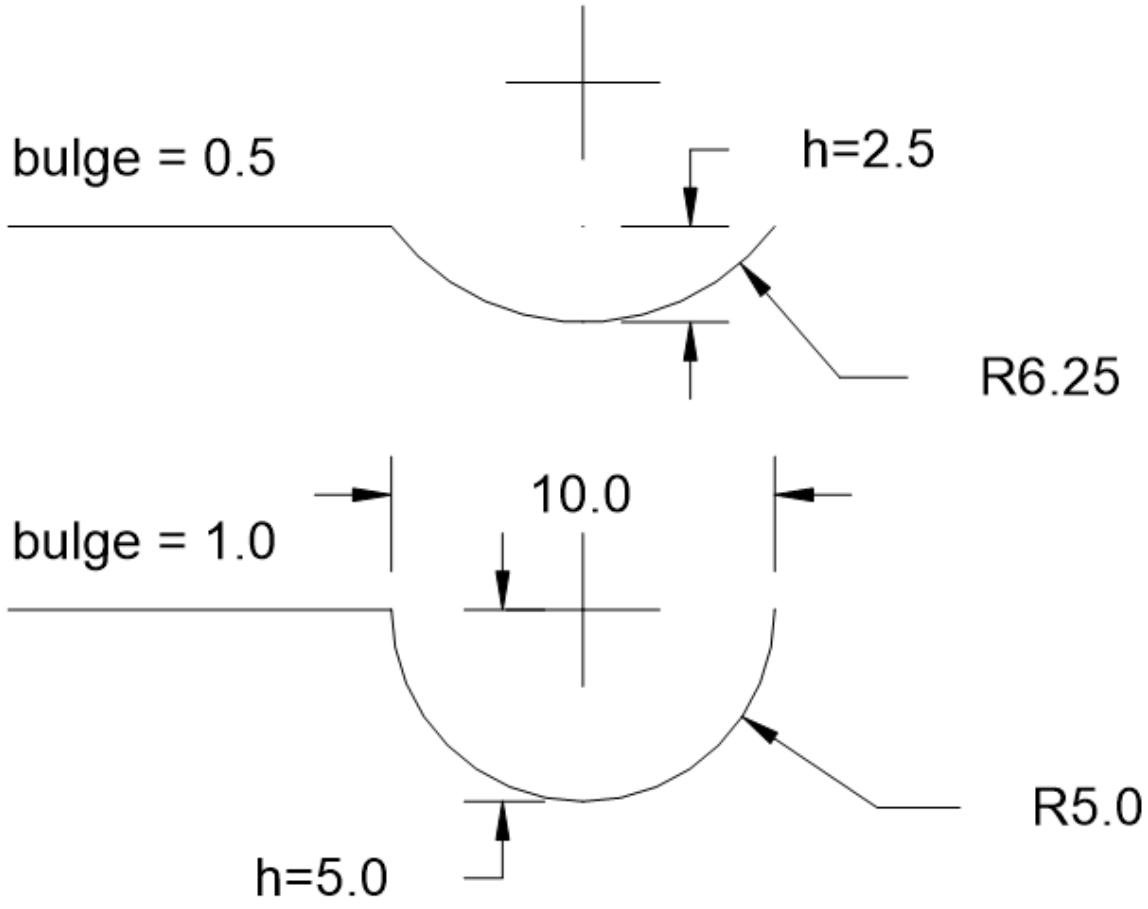
distance), a bulge value of 1 defines a semicircle. The curve is on the right side of the line for a bulge value > 0, and on the left side of the line for a bulge value < 0.

Helper functions to handle bulge values: *Bulge Related Functions*

The user defined point format, default is `xyseb`:

- `x` = x coordinate
- `y` = y coordinate
- `s` = start width
- `e` = end width
- `b` = bulge value
- `v` = (x, y) as tuple

```
msp.add_lwpolyline([(0, 0, 0), (10, 0, 1), (20, 0, 0)], format="xyb")
msp.add_lwpolyline([(0, 10, 0), (10, 10, .5), (20, 10, 0)], format="xyb")
```



6.5.10 Tutorial for Text

Add a simple one line text entity by factory function *add_text()*.

```
import ezdxf
from ezdxf.enums import TextEntityAlignment

# The TEXT entity is a DXF primitive and is supported in all DXF versions.
# The argument setup=True creates standard linetypes and text styles in the
# new DXF document.
doc = ezdxf.new("R12", setup=True)
msp = doc.modelspace()

# Use method set_placement() to define the TEXT alignment, because the
# relations between the DXF attributes 'halign', 'valign', 'insert' and
# 'align_point' are tricky.
msp.add_text("A Simple Text").set_placement(
    (2, 3),
    align=TextEntityAlignment.MIDDLE_RIGHT
)

# Using a predefined text style:
msp.add_text(
    "Text Style Example: Liberation Serif",
    height=0.35,
    dxfattribs={"style": "LiberationSerif"}
).set_placement((2, 6), align=TextEntityAlignment.LEFT)

doc.saveas("simple_text.dxf")
```

Alignments defined by the enum *TextEntityAlignment*:

Vert/Horiz	Left	Center	Right
Top	TOP_LEFT	TOP_CENTER	TOP_RIGHT
Middle	MIDDLE_LEFT	MIDDLE_CENTER	MIDDLE_RIGHT
Bottom	BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT
Baseline	LEFT	CENTER	RIGHT

Special alignments are ALIGNED and FIT, they require a second alignment point, the text is justified with the vertical alignment *Baseline* on the virtual line between these two points.

Alignment	Description
ALIGNED	Text is stretched or compressed to fit exactly between *p1* and *p2* and the text height is also adjusted to preserve height/width ratio.
FIT	Text is stretched or compressed to fit exactly between *p1* and *p2* but only the text width is adjusted, the text height is fixed by the *height* attribute.
MIDDLE	also a *special* adjustment, but the result is the same as for MIDDLE_CENTER.

Standard Text Styles

Setup some standard text styles and linetypes by argument `setup=True`:

```
doc = ezdxf.new('R12', setup=True)
```

Replaced all proprietary font declarations in `setup_styles()` (ARIAL, ARIAL_NARROW, ISOCPEUR and TIMES) by open source fonts, this is also the style name (e.g. `{'style': 'OpenSans-Italic'}`):

LiberationMono-Italic
LiberationMono-BoldItalic
LiberationMono-Bold
LiberationMono
LiberationSerif-Italic
LiberationSerif-BoldItalic
LiberationSerif-Bold
LiberationSerif
LiberationSans-Italic
LiberationSans-BoldItalic
LiberationSans-Bold
LiberationSans
OpenSansCondensed-Italic
OpenSansCondensed-Light
OpenSansCondensed-Bold
OpenSans-ExtraBoldItalic
OpenSans-ExtraBold
OpenSans-BoldItalic
OpenSans-Bold
OpenSans-SemiBoldItalic
OpenSans-SemiBold
OpenSans-Italic
OpenSans
OpenSans-Light-Italic
OpenSans-Light
STANDARD

Important: To see the defined text styles in a DXF viewer or CAD application, the applications have to know where the referenced TTF fonts can be found. This configuration is not possible by *ezdxf* and has to be done for each application as described in their documentation.

See also: *Font Resources*

New Text Style

Creating a new text style is simple:

```
doc.styles.new("myStandard", dxfattribs={"font" : "OpenSans-Regular.ttf"})
```

Getting the correct font name is often not that simple, especially on Windows. This shows the required steps to get the font name for *Open Sans*:

- open font folder *c:\windows\fonts*
- select and open the font-family *Open Sans*
- right-click on *Open Sans Standard* and select *Properties*
- on top of the first tab you see the font name: `'OpenSans-Regular.ttf'`

The style name has to be unique in the DXF document, otherwise *ezdxf* will raise an `DXFTableEntryError` exception. To replace an existing entry, delete the existing entry by `doc.styles.remove(name)`, and add the replacement entry.

3D Text

It is possible to place the 2D `Text` entity into 3D space by using the *OCS*, for further information see: *Tutorial for OCS/UCS Usage* and *Tutorial for UCS Based Transformations*.

6.5.11 Tutorial for MText and MTextEditor

The `MText` entity is a multi line entity with extended formatting possibilities and requires at least DXF version R2000, to use all features (e.g. background fill) DXF R2007 is required.

Important: The rendering result of the MTEXT entity depends on the DXF viewer or CAD application and can differ between different applications. These differences have the greatest impact on line wrapping, which can cause columns of text to have different heights in different applications!

In order for the text to look similar in different programs, the formatting should be as simple as possible or omitted altogether.

Prolog code:

```
import ezdxf

doc = ezdxf.new("R2007", setup=True)
msp = doc.modelspace()

lorem_ipsum = """
```

(continues on next page)

(continued from previous page)

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.
"""
```

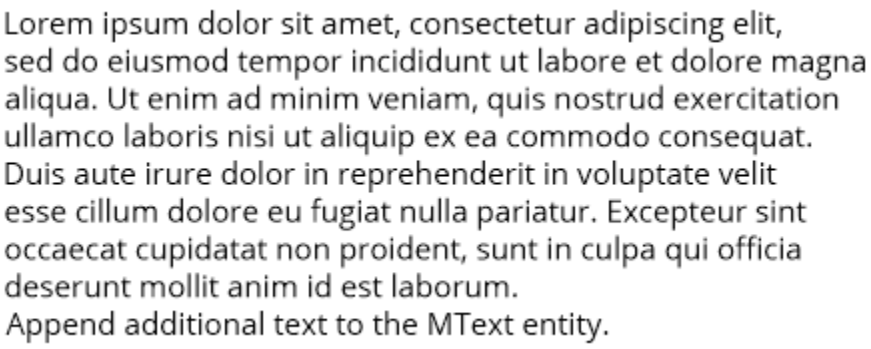
Adding a MTEXT entity

The MTEXT entity can be added to any layout (modelspace, paperspace or block) by the *add_mtext()* function.

```
# store MTEXT entity for additional manipulations
mtext = msp.add_mtext(lorem_ipsum, dxfattribs={"style": "OpenSans"})
```

This adds a MTEXT entity with text style "OpenSans". The MTEXT content can be accessed by the `text` attribute, this attribute can be edited like any Python string:

```
mtext.text += "Append additional text to the MTEXT entity."
# even shorter with __iadd__() support:
mtext += "Append additional text to the MTEXT entity."
```



The `MText` entity has an alias `MText.dxf.text` for the `MText.text` attribute for compatibility to the `Text` entity.

Important: Line endings "\n" will be replaced by the MTEXT line endings "\P" at DXF export, but **not** vice versa "\P" by "\n" at DXF file loading.

Text placement

The location of the MTEXT entity is defined by the `MText.dxf.insert` and the `MText.dxf.attachment_point` attributes in *WCS* coordinates. The `attachment_point` defines the text alignment relative to the `insert` location, default value is 1.

Attachment point constants defined in *ezdxf.lldxf.const*:

MText.dxf.attachment_point	Value
MTEXT_TOP_LEFT	1
MTEXT_TOP_CENTER	2
MTEXT_TOP_RIGHT	3
MTEXT_MIDDLE_LEFT	4
MTEXT_MIDDLE_CENTER	5
MTEXT_MIDDLE_RIGHT	6
MTEXT_BOTTOM_LEFT	7
MTEXT_BOTTOM_CENTER	8
MTEXT_BOTTOM_RIGHT	9

The MTEXT entity has a method for setting `insert`, `attachment_point` and `rotation` attributes by one call: *set_location()*

Character height

The character height is defined by the DXF attribute `MText.dxf.char_height` in drawing units, which has also consequences for the line spacing of the MTEXT entity:

```
mtext.dxf.char_height = 0.5
```

The character height can be changed inline, see also *MTEXT formatting* and *MText Inline Codes*.

Text rotation (direction)

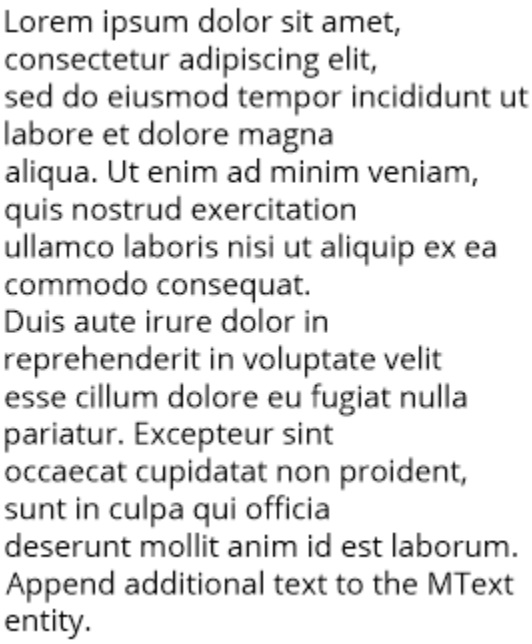
The `MText.dxf.rotation` attribute defines the text rotation as angle between the x-axis and the horizontal direction of the text in degrees. The `MText.dxf.text_direction` attribute defines the horizontal direction of MTEXT as vector in WCS. Both attributes can be present at the same entity, in this case the `MText.dxf.text_direction` attribute has the higher priority.

The MTEXT entity has two methods to get/set rotation: *get_rotation()* returns the rotation angle in degrees independent from definition as angle or direction, and *set_rotation()* set the `rotation` attribute and removes the `text_direction` attribute if present.

Defining a wrapping border

The wrapping border limits the text width and forces a line break for text beyond this border. Without attribute `dxf.width` (or setting 0) the lines are wrapped only at the regular line endings " \P" or "\n", setting the reference column width forces additional line wrappings at the given width. The text height can not be limited, the text always occupies as much space as needed.

```
mtext.dxf.width = 60
```



MTEXT formatting

MTEXT supports inline formatting by special codes: *MText Inline Codes*

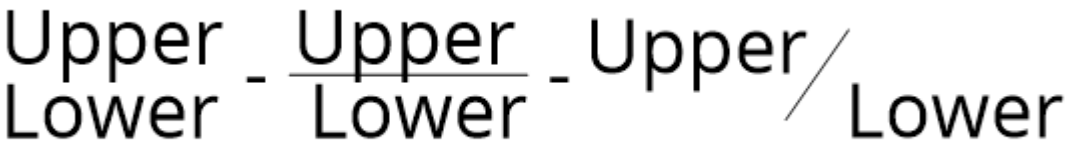
```
mtext.text = "{\\C1;red text} - {\\C3;green text} - {\\C5;blue text}"
```

See also the support class *MTextEditor*.

Stacked text

MTEXT supports stacked text:

```
# the space ' ' in front of 'Lower' and the ';' behind 'Lower' are necessary
# combined with vertical center alignment
mtext.text = "\\A1;\\SUpper^ Lower; - \\SUpper/ Lower;} - \\SUpper# Lower;"
```



See also the support class *MTextEditor*.

Background color (filling)

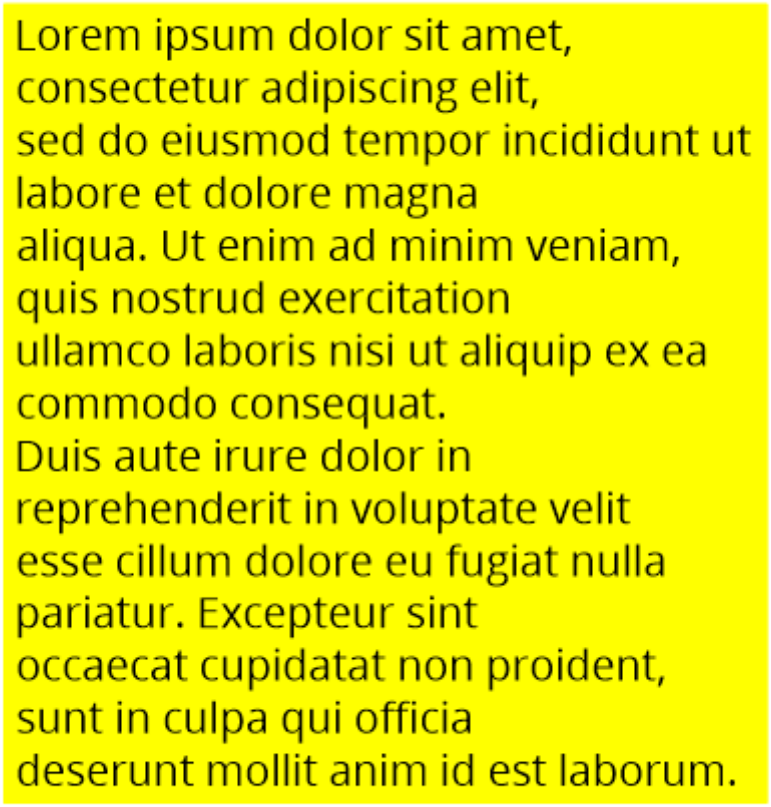
The MTEXT entity can have a background filling:

- *AutoCAD Color Index (ACI)*
- true color value as `(r, g, b)` tuple
- color name as string, use special name `'canvas'` to use the canvas background color

Because of the complex dependencies *ezdxf* provides a method to set all required DXF attributes at once:

```
mtext.set_bg_color(2, scale=1.5)
```

The parameter *scale* determines how much border there is around the text, the value is based on the text height, and should be in the range of 1 - 5, where 1 fits exact the MTEXT entity.



MTextEditor

Warning: The `MTextEditor` assembles just the inline code, which has to be parsed and rendered by the target CAD application, *ezdxf* has no influence to that result.

Keep inline formatting as simple as possible, don't test the limits of its capabilities, this will not work across different CAD applications and keep the formatting in a logic manner like, do not change paragraph properties in the middle of a paragraph.

There is no official documentation for the inline codes!

The *`MTextEditor`* class provides a floating interface to build `MText` content in an easy way.

This example only shows the connection between `MText` and the `MTextEditor`, and shows no additional features to the first example of this tutorial:

Init Editor

```
import ezdxf
from ezdxf.tools.text import MTextEditor

doc = ezdxf.new("R2007", setup=True)
msp = doc.modelspace()

lorem_ipsum = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit, ... see prolog code
"""

# create a new editor object with an initial text:
editor = MTextEditor(lorem_ipsum)

# get the MTEXT content string from the editor by the str() function:
mtext = msp.add_mtext(str(editor), dxfattribs={"style": "OpenSans"})
```

Tutorial Prolog:

```
# use constants defined in MTextEditor:
NP = MTextEditor.NEW_PARAGRAPH

ATTRIBS = {
    "char_height": 0.7,
    "style": "OpenSans",
    "width": 10,
}
editor = MTextEditor("using colors:" + NP)
```

Set Text Color

There are three ways to change the color inline:

- by color name “red”, “green”, “blue”, “yellow”, “cyan”, “magenta”, “white”
- by *AutoCAD Color Index (ACI)*
- by RGB values

```
# RED: set color by name - red, green, blue, yellow, cyan, magenta, white
editor.color("red").append("RED" + NP)
# RED: the color stays the same until the next change
editor.append("also RED" + NP)

# GREEN: change color by ACI (AutoCAD Color Index)
editor.aci(3).append("GREEN" + NP)

# BLUE: change color by RGB tuples
editor.rgb((0, 0, 255)).append("BLUE" + NP)

# add the MTEXT entity to the model space:
msp.add_mtext(str(editor), attribs)
```

Changing Text Height

The `MtextEditor.height()` method set the text height as absolute value in drawing units (text height = cap height):

```
attribs = dict(ATTRIBS)
attribs["width"] = 40.0
editor = MTextEditor("changing text height absolute: default height is 0.7" + NP)
# doubling the default height = 1.4
editor.height(1.4)
editor.append("text height: 1.4" + NP)
editor.height(3.5).append("text height: 3.5" + NP)
editor.height(0.7).append("back to default height: 0.7" + NP)
msp.add_mtext(str(editor), attribs)
```

The `MtextEditor.scale_height()` method set the text height by a relative factor, the `MtextEditor` object does not keep track of current text height, you have to do this by yourself. The initial text height is `MText.dxf.char_height`:

```
attribs = dict(ATTRIBS)
attribs["width"] = 40.0
editor = MTextEditor("changing text height relative: default height is 0.7" + NP)
# this is the default text height in the beginning:
current_height = attribs["char_height"]
# The text height can only be changed by a factor:
editor.scale_height(2)  # scale by 2 = 1.4
# keep track of the actual height:
current_height *= 2
editor.append("text height: 1.4" + NP)
# to set an absolute height, calculate the required factor:
desired_height = 3.5
factor = desired_height / current_height
editor.scale_height(factor).append("text height: 3.5" + NP)
current_height = desired_height
# and back to 0.7
editor.scale_height(0.7 / current_height).append("back to default height: 0.7" + NP)
msp.add_mtext(str(editor), attribs).set_location(insert=location)
```

Changing Font

The font name for changing `MText` fonts inline is the font family name! The font family name is the name shown in font selection widgets in desktop applications: “Arial”, “Times New Roman”, “Comic Sans MS”. The font has to be installed at the target system, else then CAD default font will be used, in AutoCAD/BricsCAD is this the font defined for the text style “Standard”.

Important: The DXF/DWG format is not optimal for preserving text layouts across multiple systems, and it’s getting really bad across different CAD applications.

```
attribs = dict(ATTRIBS)
attribs["width"] = 15.0
editor = MTextEditor("changing fonts:" + NP)
editor.append("Default: Hello World!" + NP)
editor.append("SimSun: ")
# change font in a group to revert back to the default font at the end:
simsun_editor = MTextEditor().font("SimSun").append("□□□□□" + NP)
# reverts the font back at the end of the group:
editor.group(str(simsun_editor))
# back to default font OpenSans:
editor.append("Times New Roman: ")
# change font outside of a group until next font change:
editor.font("Times New Roman").append("Привет мир!" + NP)
# If the font does not exist, a replacement font will be used:
editor.font("Does not exist").append("This is the replacement font!")
msp.add_mtext(str(editor), attribs)
```

Set Paragraph Properties

The paragraph properties are set by the *paragraph()* method and a *ParagraphProperties* object, which bundles all paragraph properties in a named tuple.

Each paragraph can have its own properties for:

- indentation arguments:
  - `indent` is the left indentation of the first line
  - `left` is the left side indentation of the paragraph
  - `right` is the right side indentation of the paragraph
- text adjustment: `align`, by enum `MTextParagraphAlignment`
  - MTextParagraphAlignment.LEFT
  - MTextParagraphAlignment.RIGHT
  - MTextParagraphAlignment.CENTER
  - MTextParagraphAlignment.JUSTIFIED
  - MTextParagraphAlignment.DISTRIBUTED
- tabulator stops: `tab_stops`, a tuple of tabulator stops

Indentation and tabulator stops are multiples of the default `MText` text height stored in `MText.dxf.char_height`. Calculate the drawing units for indentation and tabulator stops, by multiplying the indentation value by the `char_height` value.

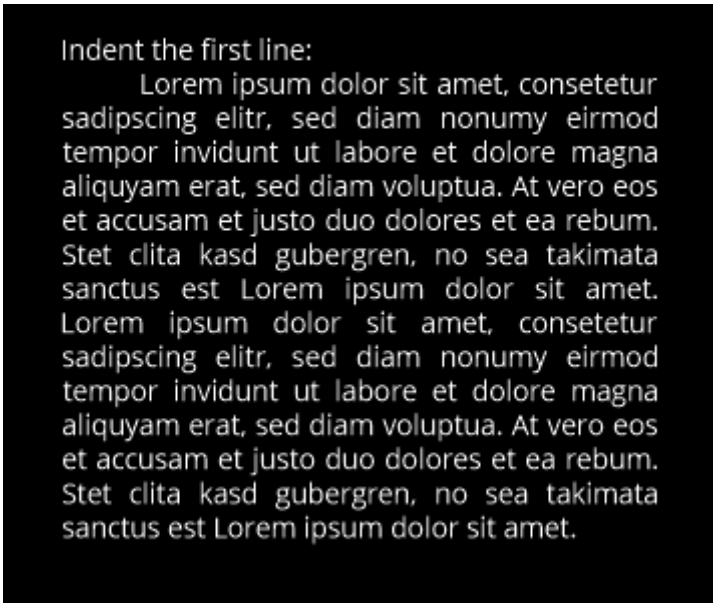
`Mtext` paragraphs are separated by new paragraph "\P" characters.

```
# import support classes:
from ezdxf.tools.text import ParagraphProperties, MTextParagraphAlignment

attribs = dict(ATTRIBS)
```

(continues on next page)

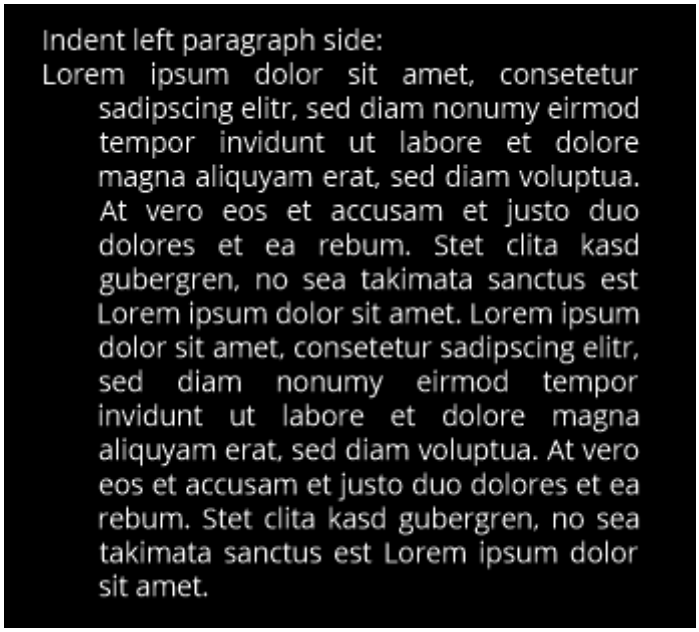
(continued from previous page)

```
attribs["char_height"] = 0.25
attribs["width"] = 7.5
editor = MTextEditor("Indent the first line:" + NP)
props = ParagraphProperties(
    indent=1,  # indent first line = 1x0.25 drawing units
    align=MTextParagraphAlignment.JUSTIFIED
)
editor.paragraph(props)
editor.append(lorem_ipsum)
msp.add_mtext(str(editor), attribs)
```



The first line indentation “indent” is relative to the “left” indentation.

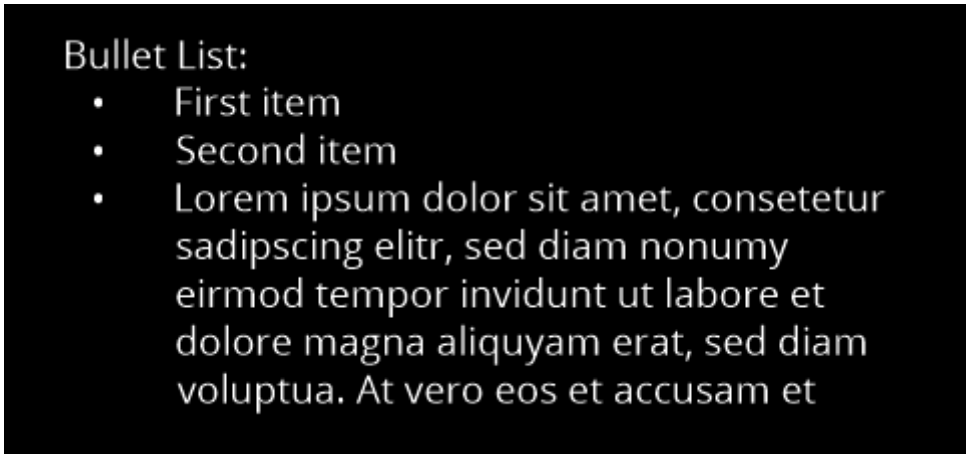
```
# import support classes:
from ezdxf.tools.text import ParagraphProperties, MTextParagraphAlignment

attribs = dict(ATTRIBS)
attribs["char_height"] = 0.25
attribs["width"] = 7.5
editor = MTextEditor("Indent left paragraph side:" + NP)
indent = 0.7  # 0.7 * 0.25 = 0.175 drawing units
props = ParagraphProperties(
    # first line indentation is relative to "left", this reverses the
    # left indentation:
    indent=-indent,  # first line
    # indent left paragraph side:
    left=indent,
    align=MTextParagraphAlignment.JUSTIFIED
)
editor.paragraph(props)
editor.append(" ".join(lorem_ipsum(100)))
msp.add_mtext(str(editor), attribs).set_location(insert=location)
```



Bullet List

There are no special commands to build bullet list, the list is build of indentation and a tabulator stop. Each list item needs a marker as an arbitrary string. For more information about paragraph indentation and tabulator stops see also chapter *Set Paragraph Properties*.

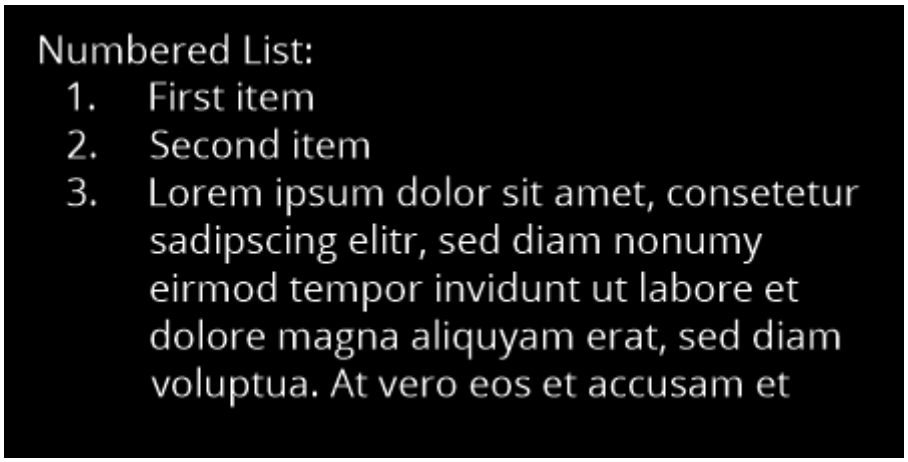
```
attribs = dict(ATTRIBS)
attribs["char_height"] = 0.25
attribs["width"] = 7.5
bullet = "•"  # alt + numpad 7
editor = MTextEditor("Bullet List:" + NP)
editor.bullet_list(
    indent=1,
    bullets=[bullet] * 3,  # each list item needs a marker
    content=[
        "First item",
        "Second item",
        " ".join(lorem_ipsum(30)),
    ])
msp.add_mtext(str(editor), attribs)
```



Numbered List

There are no special commands to build numbered list, the list is build of indentation and a tabulator stop. There is no automatic numbering, but therefore the absolute freedom for using any string as list marker. For more information about paragraph indentation and tabulator stops see also chapter *Set Paragraph Properties*.

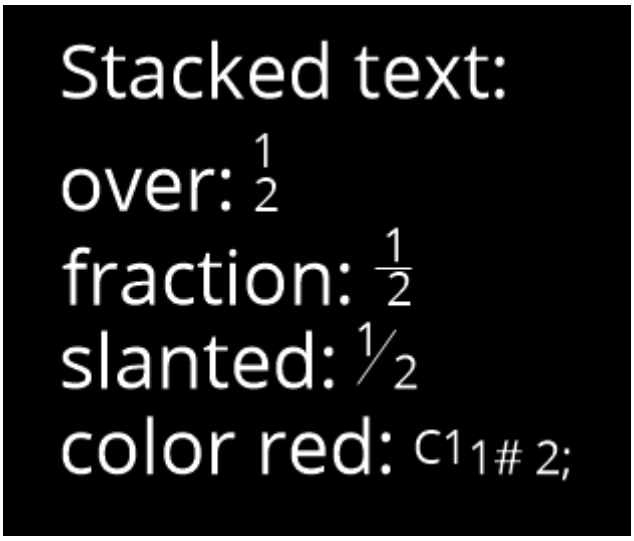
```
attribs = dict(ATTRIBS)
attribs["char_height"] = 0.25
attribs["width"] = 7.5
editor = MTextEditor("Numbered List:" + NP)
editor.bullet_list(
    indent=1,
    bullets=["1.", "2.", "3."],
    content=[
        "First item",
        "Second item",
        " ".join(lorem_ipsum(30)),
    ])
msp.add_mtext(str(editor), attribs)
```



Stacked Text

`MText` supports stacked text (fractions) as a single inline code, which means it is not possible to change any property inside the fraction. This example shows a fraction with scaled down text height, placed in a group to revert the text height afterwards:

```
editor = MTextEditor("Stacked text:" + NP)

stack = MTextEditor().scale_height(0.6).stack("1", "2", "^")
editor.append("over: ").group(str(stack)).append(NP)

stack = MTextEditor().scale_height(0.6).stack("1", "2", "/")
editor.append("fraction: ").group(str(stack)).append(NP)

stack = MTextEditor().scale_height(0.6).stack("1", "2", "#")
editor.append("slanted: ").group(str(stack)).append(NP)

# Additional formatting in numerator and denominator is not supported
# by AutoCAD or BricsCAD, switching the color inside the stacked text
# to red does not work:
numerator = MTextEditor().color("red").append("1")
stack = MTextEditor().scale_height(0.6).stack(str(numerator), "2", "#")
editor.append("color red: ").group(str(stack)).append(NP)

msp.add_mtext(str(editor), attribs)
```



See also:

- *MTextEditor* example code on github.
- Documentation of *MTextEditor*

6.5.12 Tutorial for Spline

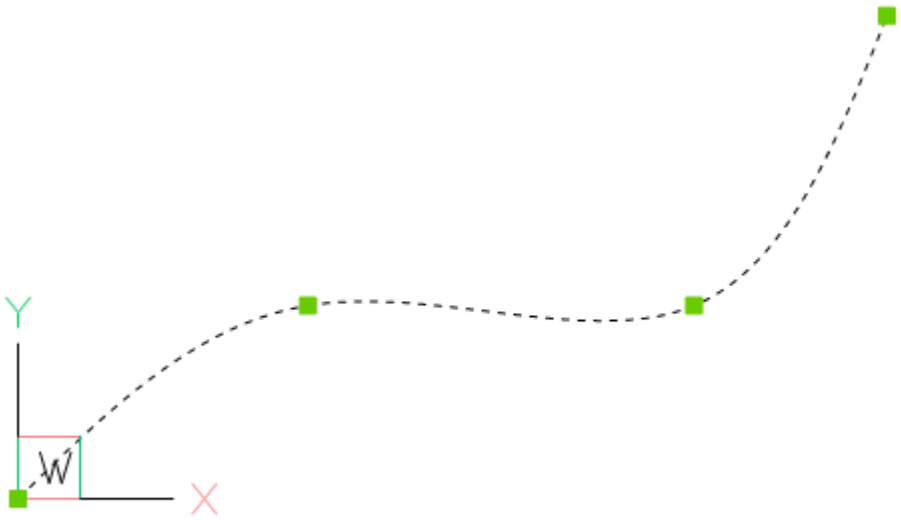
Background information about B-spline at Wikipedia.

Splines from fit points

Splines can be defined by fit points only, this means the curve passes all given fit points. AutoCAD and BricsCAD generates required control points and knot values by itself, if only fit points are present.

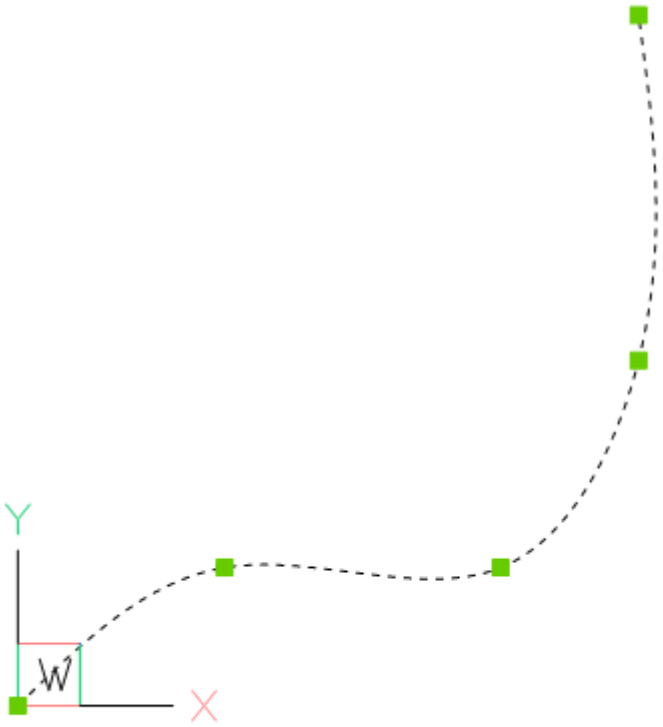
Create a simple spline:

```
doc = ezdxf.new("R2000")

fit_points = [(0, 0, 0), (750, 500, 0), (1750, 500, 0), (2250, 1250, 0)]
msp = doc.modelspace()
spline = msp.add_spline(fit_points)
```



Append a fit point to a spline:

```
# fit_points, control_points, knots and weights are list-like containers:
spline.fit_points.append((2250, 2500, 0))
```



You can set additional *control points*, but if they do not fit the auto-generated AutoCAD values, they will be ignored and don't mess around with knot values.

```
doc = ezdxf.readfile("AutoCAD_generated.dxf")

msp = doc.modelspace()
spline = msp.query("SPLINE").first

# fit_points, control_points, knots and weights are list-like objects:
spline.fit_points.append((2250, 2500, 0))
```

As far as I have tested, this approach works without complaints from AutoCAD, but for the case of problems remove invalid data from the SPLINE entity:

```
# current control points do not match spline defined by fit points
spline.control_points = []

# count of knots is not correct:
# count of knots = count of control points + degree + 1
spline.knots = []

# same for weights, count of weights == count of control points
spline.weights = []
```

Splines by control points

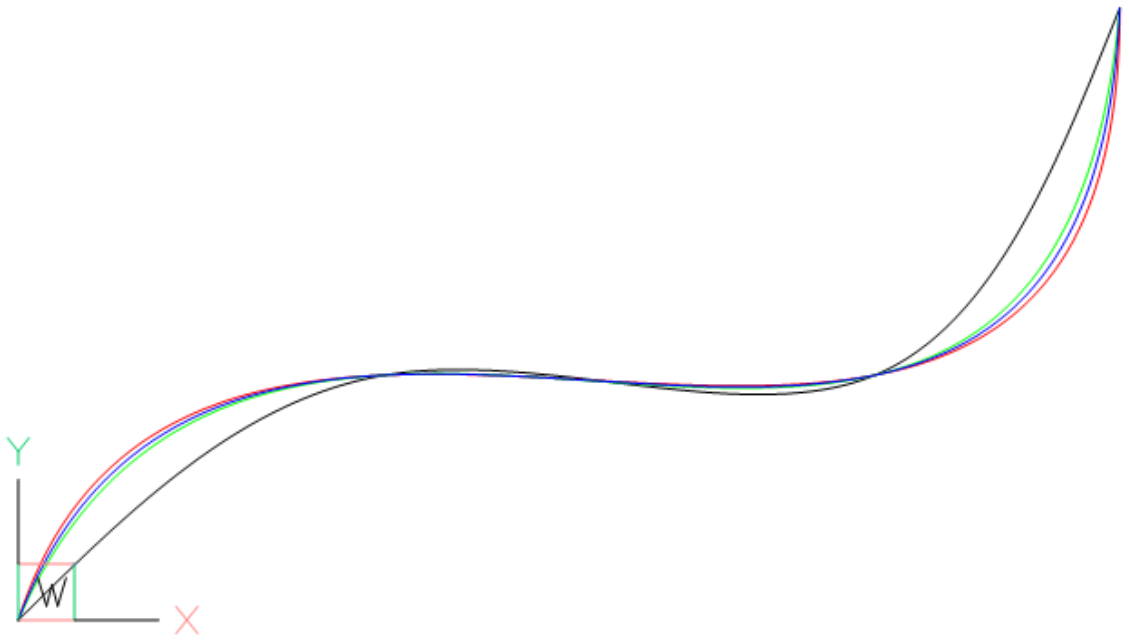
Creating splines from fit points is the easiest way, but this method is also the least accurate, because a spline is defined by control points and knot values, which are generated for the case of a definition by fit points, and the worst fact is that for every given set of fit points exist an infinite number of possible splines as solution.

AutoCAD (and BricsCAD) uses an unknown proprietary algorithm to generate control points and knot values from fit points. Therefore splines generated from fit points by *ezdxf* do not match splines generated by AutoCAD (BricsCAD).

To ensure the same spline geometry for all CAD applications, the spline has to be defined by control points. The method *add_spline_control_frame()* adds a spline passing the given fit points by calculating the control points by the Global Curve Interpolation algorithm. There is also a low level function *ezdxf.math.global_bspline_interpolation()* which calculates the control points from fit points.

```
msp.add_spline_control_frame(fit_points, method='uniform', dxfattribs={'color': 1})
msp.add_spline_control_frame(fit_points, method='chord', dxfattribs={'color': 3})
msp.add_spline_control_frame(fit_points, method='centripetal', dxfattribs={'color': 5}
↪)
```

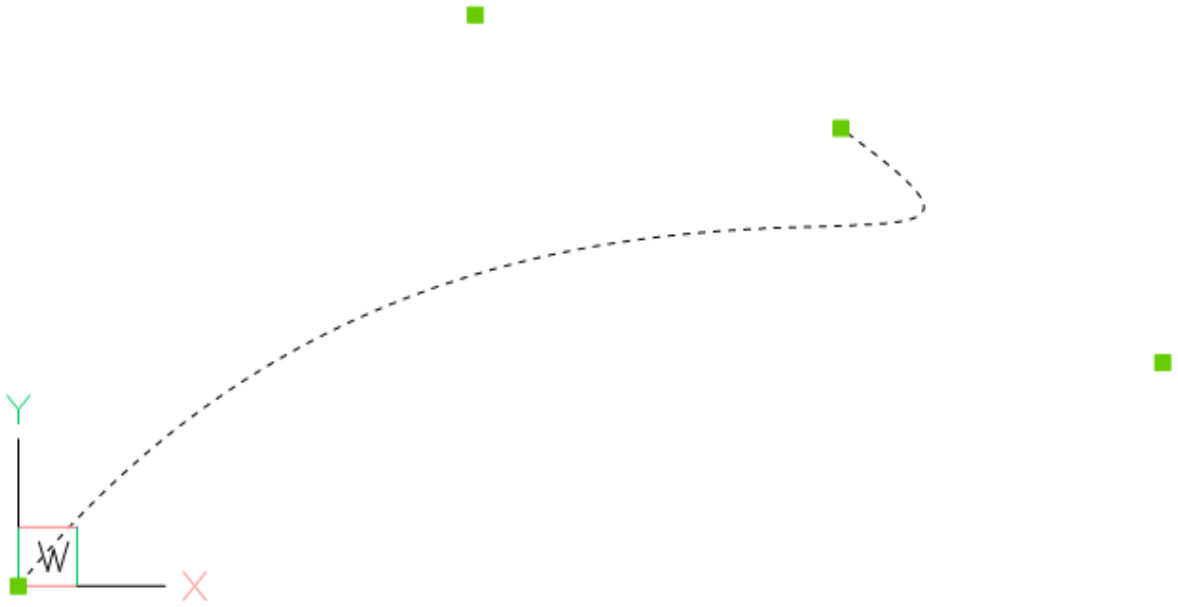
- black curve: AutoCAD/BricsCAD spline generated from fit points
- red curve: spline curve interpolation, “uniform” method
- green curve: spline curve interpolation, “chord” method
- blue curve: spline curve interpolation, “centripetal” method



Open Spline

Add and open (clamped) spline defined by control points with the method *add_open_spline()*. If no knot values are given, an open uniform knot vector will be generated. A clamped B-spline starts at the first control point and ends at the last control point.

```
control_points = [(0, 0, 0), (1250, 1560, 0), (3130, 610, 0), (2250, 1250, 0)]
msp.add_open_spline(control_points)
```

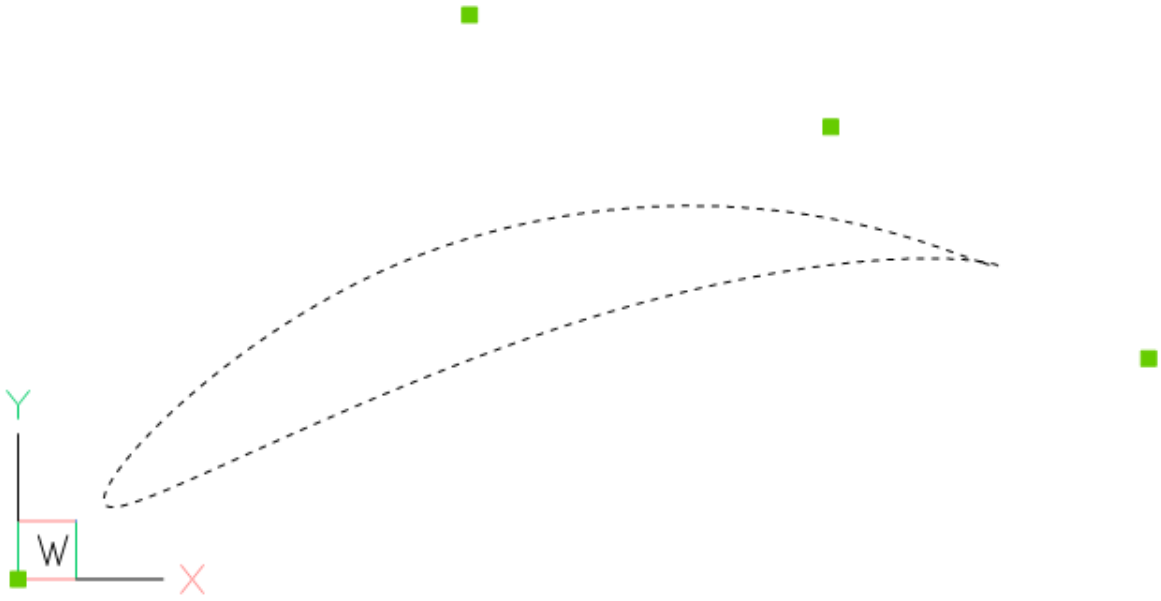


Rational Spline

Rational B-splines have a weight for every control point, which can raise or lower the influence of the control point, default weight = `1`, to lower the influence set a weight < `1` to raise the influence set a weight > `1`. The count of weights has to be always equal to the count of control points.

Example to raise the influence of the first control point:

```
msp.add_closed_rational_spline(control_points, weights=[3, 1, 1, 1])
```



Spline properties

Check if spline is a closed curve or close/open spline, for a closed spline the last point is connected to the first point:

```
if spline.closed:
    # this spline is closed
    pass

# close spline
spline.closed = True

# open spline
spline.closed = False
```

Set start- and end tangent for splines defined by fit points:

```
spline.dxf.start_tangent = (0, 1, 0)
spline.dxf.end_tangent = (1, 0, 0)
```

Get data count as stored in DXF attributes:

```
count = spline.dxf.n_fit_points
count = spline.dxf.n_control_points
count = spline.dxf.n_knots
```

Get data count from existing data:

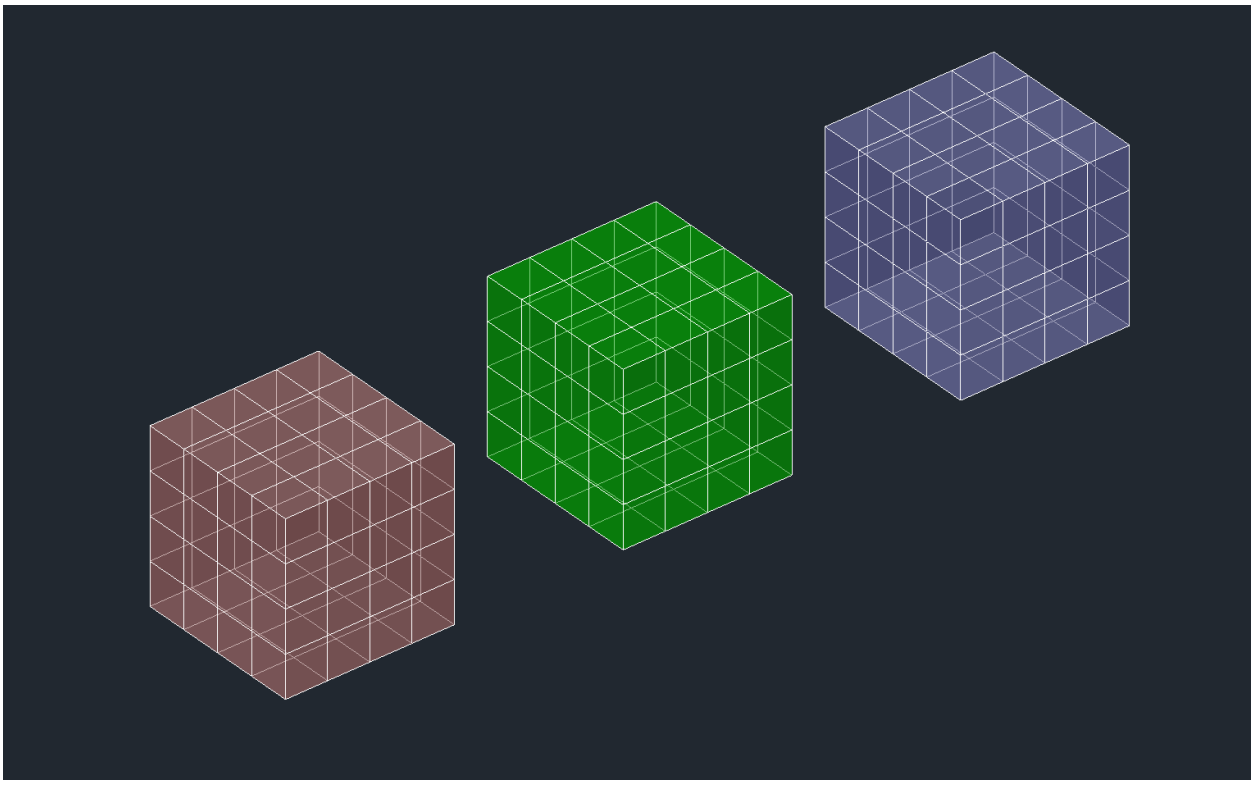
```
count = spline.fit_point_count
count = spline.control_point_count
count = spline.knot_count
```

6.5.13 Tutorial for Polyface

The *Polyface* entity represents a 3D mesh build of vertices and faces and is just an extended POLYLINE entity with a complex VERTEX structure. The Polyface entity was used in DXF R12 and older DXF versions and is still supported by newer DXF versions. The new *Mesh* entity stores the same data much more efficient but requires DXF R2000 or newer. The Polyface entity supports only triangles and quadrilaterals as faces, the Mesh entity supports also n-gons.

Its recommended to use the *MeshBuilder* objects to create 3D meshes and render them as POLYFACE entities by the render_polymesh() method into a layout:

```
import ezdxf
from ezdxf import colors
from ezdxf.gfxattribs import GfxAttribs
from ezdxf.render import forms

cube = forms.cube().scale_uniform(10).subdivide(2)
red = GfxAttribs(color=colors.RED)
green = GfxAttribs(color=colors.GREEN)
blue = GfxAttribs(color=colors.BLUE)

doc = ezdxf.new()
msp = doc.modelspace()

# render as MESH entity
cube.render_mesh(msp, dxfattribs=red)
cube.translate(20)

# render as POLYFACE a.k.a. POLYLINE entity
cube.render_polyface(msp, dxfattribs=green)
cube.translate(20)

# render as a bunch of 3DFACE entities
cube.render_3dfaces(msp, dxfattribs=blue)

doc.saveas("meshes.dxf")
```



Warning: If the mesh contains n-gons the render methods for POLYFACE and 3DFACES subdivides the n-gons into triangles, which does **not** work for concave faces.

The usage of the *MeshBuilder* object is also recommended for inspecting `Polyface` entities:

- `MeshBuilder.vertices` is a sequence of 3D points as *ezdxf.math.Vec3* objects
- a face in `MeshBuilder.faces` is a sequence of indices into the `MeshBuilder.vertices` sequence

```
import ezdxf
from ezdxf.render import MeshBuilder

def process(mesh):
    # vertices is a sequence of 3D points
    vertices = mses.vertices
    # a face is a sequence of indices into the vertices sequence
    faces = mesh.faces
    ...

doc = ezdxf.readfile("meshes.dxf")
msp = doc.modelspace()
for polyline in msp.query("POLYLINE"):
    if polyline.is_poly_face_mesh:
        mesh = MeshBuilder.from_polyface(polyline)
        process(mesh)
```

See also:

Tutorial for Mesh

6.5.14 Tutorial for Mesh

The *Mesh* entity is a 3D object in *WCS* build up from vertices and faces.

Create a cube mesh by directly accessing the base data structures:

```
import ezdxf


# 8 corner vertices
cube_vertices = [
    (0, 0, 0),
    (1, 0, 0),
    (1, 1, 0),
    (0, 1, 0),
    (0, 0, 1),
    (1, 0, 1),
    (1, 1, 1),
    (0, 1, 1),
]

# 6 cube faces
cube_faces = [
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [0, 1, 5, 4],
    [1, 2, 6, 5],
    [3, 2, 6, 7],
    [0, 3, 7, 4]
]

# MESH requires DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()
mesh = msp.add_mesh()
# do not subdivide cube, 0 is the default value
mesh.dxf.subdivision_levels = 0
with mesh.edit_data() as mesh_data:
    mesh_data.vertices = cube_vertices
    mesh_data.faces = cube_faces

doc.saveas("cube_mesh_1.dxf")
```

Create a cube mesh by assembling single faces using the *edit_data()* context manager of the *Mesh* class and the helper class *MeshData*:

```
import ezdxf


# 8 corner vertices
p = [
    (0, 0, 0),
    (1, 0, 0),
    (1, 1, 0),
    (0, 1, 0),
    (0, 0, 1),
    (1, 0, 1),
    (1, 1, 1),
```

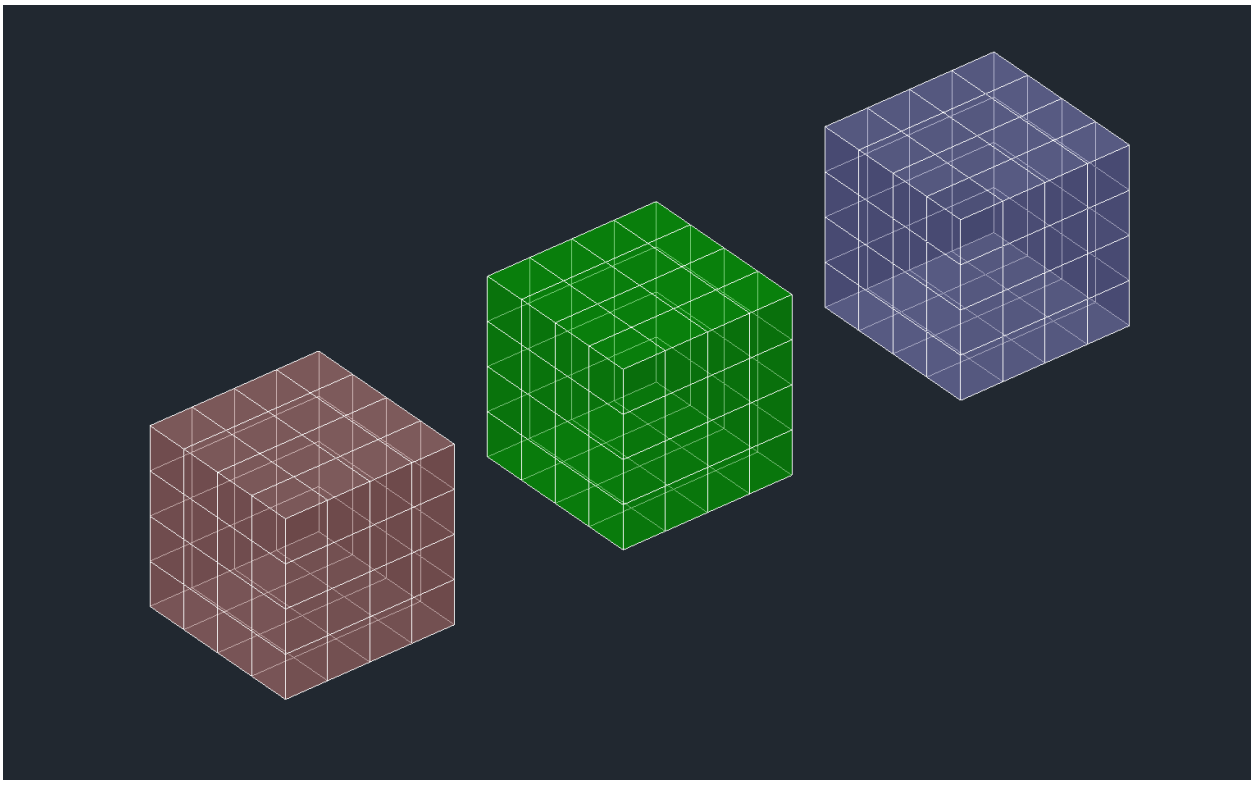
(continues on next page)

(continued from previous page)

```
    (0, 1, 1),
]

# MESH requires DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()
mesh = msp.add_mesh()

with mesh.edit_data() as mesh_data:
    mesh_data.add_face([p[0], p[1], p[2], p[3]])
    mesh_data.add_face([p[4], p[5], p[6], p[7]])
    mesh_data.add_face([p[0], p[1], p[5], p[4]])
    mesh_data.add_face([p[1], p[2], p[6], p[5]])
    mesh_data.add_face([p[3], p[2], p[6], p[7]])
    mesh_data.add_face([p[0], p[3], p[7], p[4]])
    # optional call optimize(): minimizes the vertex count
    mesh_data.optimize()

doc.saveas("cube_mesh_2.dxf")
```

Its recommended to use the *MeshBuilder* objects to create 3D meshes and render them as MESH entities by the *render_mesh()* method into a layout:

```
import ezdxf
from ezdxf import colors
from ezdxf.gfxattribs import GfxAttribs
from ezdxf.render import forms

cube = forms.cube().scale_uniform(10).subdivide(2)
red = GfxAttribs(color=colors.RED)
green = GfxAttribs(color=colors.GREEN)
blue = GfxAttribs(color=colors.BLUE)

doc = ezdxf.new()
msp = doc.modelspace()

# render as MESH entity
cube.render_mesh(msp, dxfattribs=red)
cube.translate(20)

# render as POLYFACE a.k.a. POLYLINE entity
cube.render_polyface(msp, dxfattribs=green)
cube.translate(20)

# render as a bunch of 3DFACE entities
cube.render_3dfaces(msp, dxfattribs=blue)

doc.saveas("meshes.dxf")
```



There exist some tools to manage meshes:

- *ezdxf.render.MeshBuilder*: The *MeshBuilder* classes are helper tools to manage meshes buildup by vertices and faces.
- *ezdxf.render.MeshTransformer*: Same functionality as *MeshBuilder* but supports inplace transformation.
- *ezdxf.render.MeshDiagnose*: A diagnose tool which can be used to analyze and detect errors of *MeshBuilder* objects like topology errors for closed surfaces.
- *ezdxf.render.FaceOrientationDetector*: A helper class for face orientation and face normal vector detection

The *ezdxf.render.forms* module provides function to create basic geometries like cube, cone, sphere and so on and functions to create meshes from profiles by extrusion, rotation or sweeping.

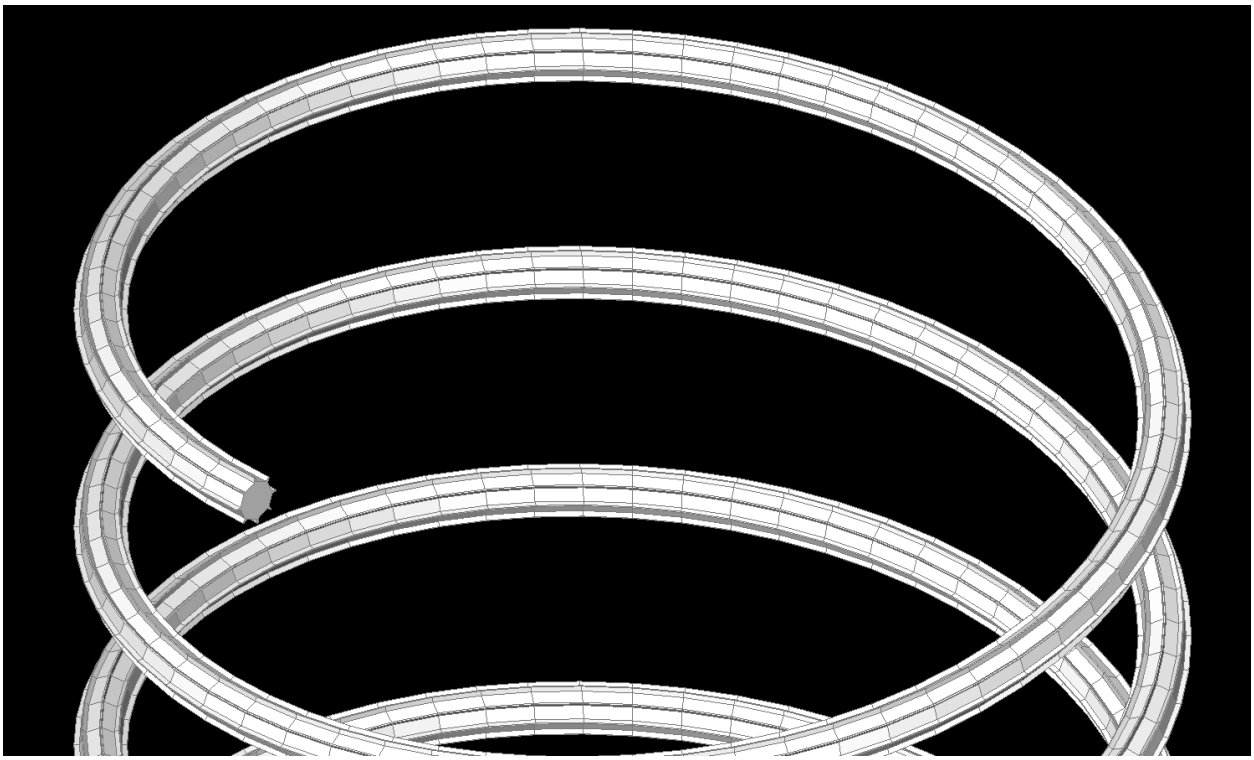
This example shows how to sweep a gear profile along a helix:

```
import ezdxf
from ezdxf.render import forms

doc = ezdxf.new()
doc.layers.add("MESH", color=ezdxf.colors.YELLOW)
msp = doc.modelspace()
# sweeping a gear-profile
gear = forms.gear(
    8, top_width=0.01, bottom_width=0.02, height=0.02, outside_radius=0.1
)
helix = path.helix(radius=2, pitch=1, turns=6)
# along a helix spine
sweeping_path = helix.flattening(0.1)
mesh = forms.sweep(gear, sweeping_path, close=True, caps=True)
```

(continues on next page)

(continued from previous page)

```
# and render as MESH entity
mesh.render_mesh(msp, dxfattribs={"layer": "MESH"})
doc.saveas("gear_along_helix.dxf")
```



6.5.15 Tutorial for Hatch

Create hatches with one boundary path

The simplest form of the *Hatch* entity has one polyline path with only straight lines as boundary path:

```
import ezdxf

# hatch requires DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is a 2D element
# vertex format for the polyline path is: (x, y[, bulge])
# there are no bulge values in this example
hatch.paths.add_polyline_path(
    [(0, 0), (10, 0), (10, 10), (0, 10)], is_closed=True
)

doc.saveas("solid_hatch_polyline_path.dxf")
```

But like all polyline entities the polyline path can also have bulge values:

```
import ezdxf

# hatch requires the DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is a 2D element
# vertex format for the polyline path is: (x, y[, bulge])
# bulge value 1 = an arc with diameter=10 (= distance to next vertex * bulge value)
# bulge value > 0 ... arc is right of line
# bulge value < 0 ... arc is left of line
hatch.paths.add_polyline_path(
    [(0, 0, 1), (10, 0), (10, 10, -0.5), (0, 10)], is_closed=True
)

doc.saveas("solid_hatch_polyline_path_with_bulge.dxf")
```

The most flexible way to define a boundary path is the edge path. An edge path can have multiple edges and each edge can be one of the following elements:

- line `EdgePath.add_line()`
- arc `EdgePath.add_arc()`
- ellipse `EdgePath.add_ellipse()`
- spline `EdgePath.add_spline()`

Create a solid hatch with an edge path (ellipse) as boundary path:

```
import ezdxf

# hatch requires the DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# important: major axis >= minor axis (ratio <= 1.)
# minor axis length = major axis length * ratio
msp.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5)

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is a 2D element
edge_path = hatch.paths.add_edge_path()
# each edge path can contain line, arc, ellipse and spline elements
# important: major axis >= minor axis (ratio <= 1.)
edge_path.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5)

doc.saveas("solid_hatch_ellipse.dxf")
```

Create hatches with multiple boundary paths (islands)

The DXF attribute `hatch_style` defines the island detection style:

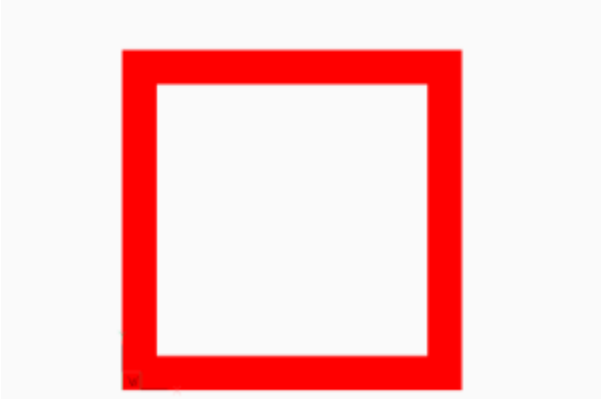
0	nested - altering filled and unfilled areas
1	outer - area between *external* and *outermost* path is filled
2	ignore - *external* path is filled

```
hatch = msp.add_hatch(
    color=1,
    dxfattribs={
        "hatch_style": ezdxf.const.HATCH_STYLE_NESTED,
        # 0 = nested: ezdxf.const.HATCH_STYLE_NESTED
        # 1 = outer: ezdxf.const.HATCH_STYLE_OUTERMOST
        # 2 = ignore: ezdxf.const.HATCH_STYLE_IGNORE
    },
)

# The first path has to set flag: 1 = external
# flag const.BOUNDARY_PATH_POLYLINE is added (OR) automatically
hatch.paths.add_polyline_path(
    [(0, 0), (10, 0), (10, 10), (0, 10)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL,
)
```

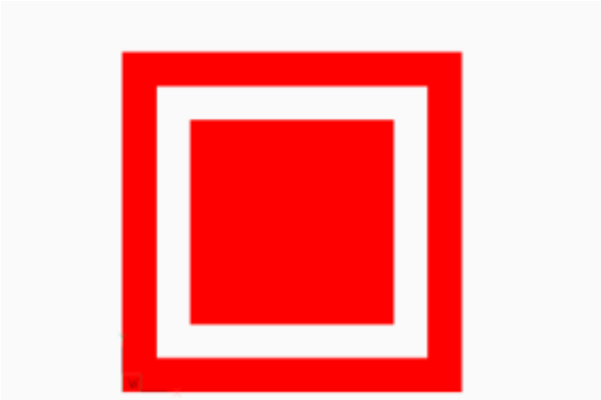
This is also the result for all 4 paths and `hatch_style` set to `2` (ignore).

```
# The second path has to set flag: 16 = outermost
hatch.paths.add_polyline_path(
    [(1, 1), (9, 1), (9, 9), (1, 9)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_OUTERMOST,
)
```

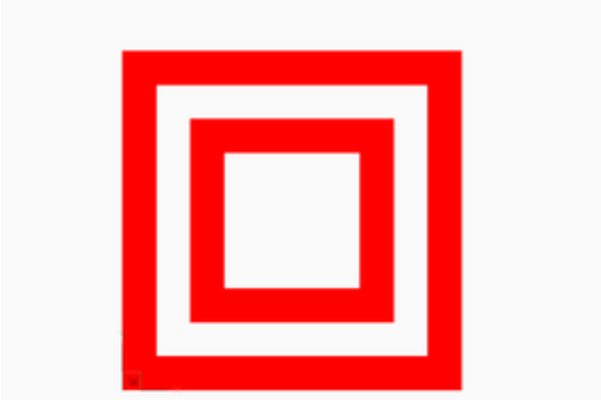
This is also the result for all 4 paths and `hatch_style` set to `1` (outer).



```
# The third path has to set flag: 0 = default
hatch.paths.add_polyline_path(
    [(2, 2), (8, 2), (8, 8), (2, 8)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_DEFAULT,
)
```



```
# The forth path has to set flag: 0 = default, and so on
hatch.paths.add_polyline_path(
    [(3, 3), (7, 3), (7, 7), (3, 7)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_DEFAULT,
)

doc.saveas(OUTDIR / "solid_hatch_islands_04.dxf")
```



The expected result of combinations of various `hatch_style` values and paths *flags*, or the handling of overlapping paths is not documented by the DXF reference, so don't ask me, ask Autodesk or just try it by yourself and post your experience in the forum.

Example for Edge Path Boundary

```
hatch = msp.add_hatch(color=1)

# 1. polyline path
hatch.paths.add_polyline_path(
    [
        (240, 210, 0),
        (0, 210, 0),
        (0, 0, 0.0),
        (240, 0, 0),
    ],
    is_closed=1,
    flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL,
)
# 2. edge path
edge_path = hatch.paths.add_edge_path(flags=ezdxf.const.BOUNDARY_PATH_OUTERMOST)
edge_path.add_spline(
    control_points=[
        (126.658105895725, 177.0823706957212),
        (141.5497003747484, 187.8907860433995),
        (205.8997365206943, 154.7946313459515),
        (113.0168862297068, 117.8189380884978),
        (202.9816918983783, 63.17222935389572),
        (157.363511042264, 26.4621294342132),
        (144.8204003260554, 28.4383294369643),
    ],
    knot_values=[
        0.0,
        0.0,
        0.0,
        0.0,
        55.20174685732758,
        98.33239645153571,
        175.1126541251052,
        213.2061566683142,
        213.2061566683142,
        213.2061566683142,
        213.2061566683142,
    ],
)
edge_path.add_arc(
    center=(152.6378550678883, 128.3209356351659),
    radius=100.1880612627354,
    start_angle=94.4752130054052,
    end_angle=177.1345242028005,
)
edge_path.add_line(
    (52.57506282464041, 123.3124200796114),
    (126.658105895725, 177.0823706957212),
)
```

Associative Boundary Paths

A HATCH entity can be associative to a base geometry, which means if the base geometry is edited in a CAD application the HATCH get the same modification. Because *ezdxf* is **not** a CAD application, this association is **not** maintained nor verified by *ezdxf*, so if you modify the base geometry afterwards the geometry of the boundary path is not updated and no verification is done to check if the associated geometry matches the boundary path, this opens many possibilities to create invalid DXF files: USE WITH CARE.

This example associates a LWPOLYLINE entity to the hatch created from the LWPOLYLINE vertices:

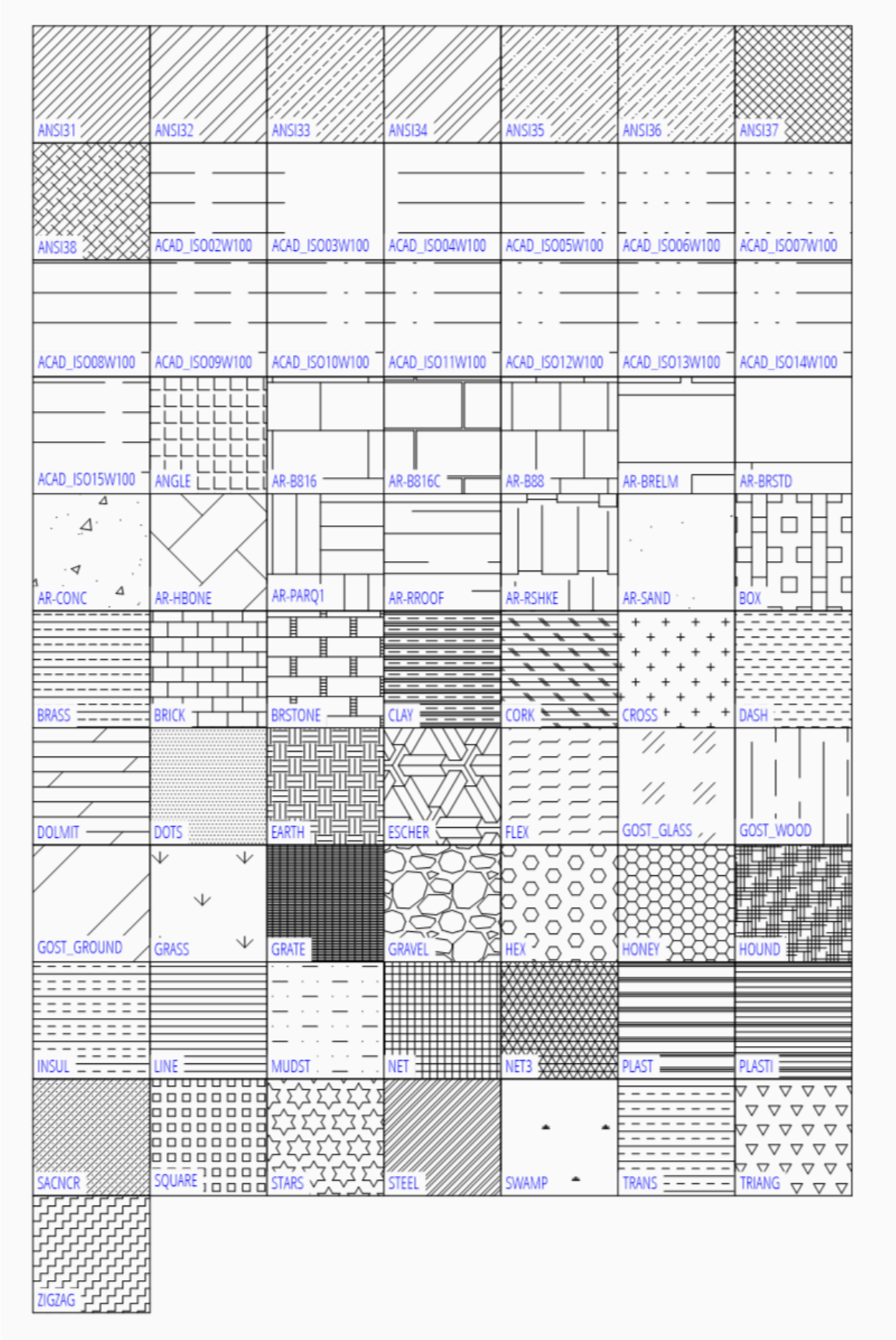
```
# Create base geometry
lwpolyline = msp.add_lwpolyline(
    [(0, 0, 0), (10, 0, 0.5), (10, 10, 0), (0, 10, 0)],
    format="xyb",
    close=True,
)

hatch = msp.add_hatch(color=1)
path = hatch.paths.add_polyline_path(
    # get path vertices from associated LWPOLYLINE entity
    lwpolyline.get_points(format="xyb"),
    # get closed state also from associated LWPOLYLINE entity
    is_closed=lwpolyline.closed,
)

# Set association between boundary path and LWPOLYLINE
hatch.associate(path, [lwpolyline])
```

An `EdgePath` needs associations to all geometry entities forming the boundary path.

Predefined Hatch Pattern

Use predefined hatch pattern by name:

```
hatch.set_pattern_fill("ANSI31", scale=0.5)
```


ANSI31
ANSI32
ANSI33
ANSI34
ANSI35
ANSI36
ANSI37
ANSI38
ACAD_ISO02W100
ACAD_ISO03W100
ACAD_ISO04W100
ACAD_ISO05W100
ACAD_ISO06W100
ACAD_ISO07W100
ACAD_ISO08W100
ACAD_ISO09W100
ACAD_ISO10W100
ACAD_ISO11W100
ACAD_ISO12W100
ACAD_ISO13W100
ACAD_ISO14W100
ACAD_ISO15W100
ANGLE
AR-B816
AR-B816C
AR-B88
AR-BRELM
AR-BRSTD
AR-CONC
AR-HBONE
AR-PARQ1
AR-RROOF
AR-RSHKE
AR-SAND
BOX
BRASS
BRICK
BRSTONE
CLAY
CORK
CROSS
DASH
DOLMIT
DOTS
EARTH
ESCHER
FLEX
GOST_GLASS
GOST_WOOD
GOST_GROUND
GRASS
GRATE
GRAVEL
HEX
HONEY
HOUND
INSUL
LINE
MUDST
NET
NET3
PLAST
PLASTI
SACNCR
SQUARE
STARS
STEEL
SWAMP
TRANS
TRIANG
ZIGZAG

See also:

Tutorial for Hatch Pattern Definition

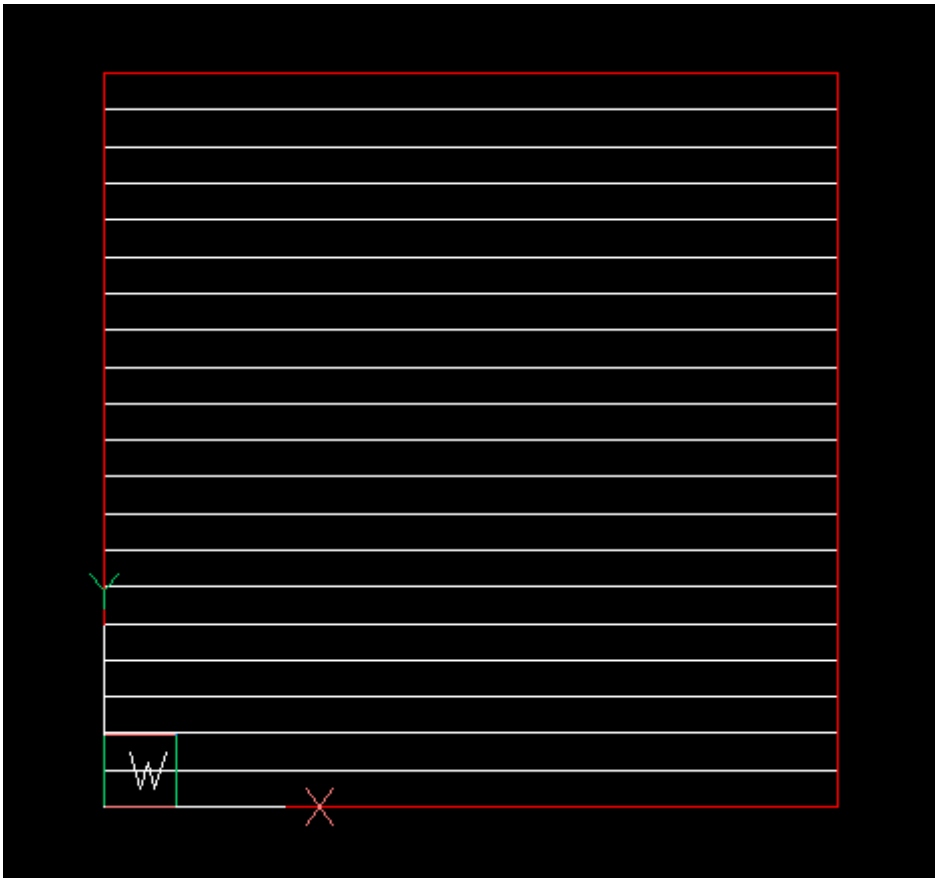
6.5.16 Tutorial for Hatch Pattern Definition

A hatch pattern consist of one or more hatch lines. A hatch line defines a set of lines which have the same orientation an the same line pattern. All the lines defined by a hatch line are parallel and have a constant distance to each other. The *origin* defines the start point of the hatch line and also the starting point of the line pattern. The *direction* defines the angle between the *WCS* x-axis and the hatch line. The *offset* is a 2D vector which will be added consecutively the the origin for each new hatch line. The line pattern has the same format as as the simple linetype pattern (*Tutorial for Creating Linetype Pattern*).

Important: The hatch pattern must be defined for a hatch scaling factor of 1.0 and a hatch rotation angle of 0 degrees!

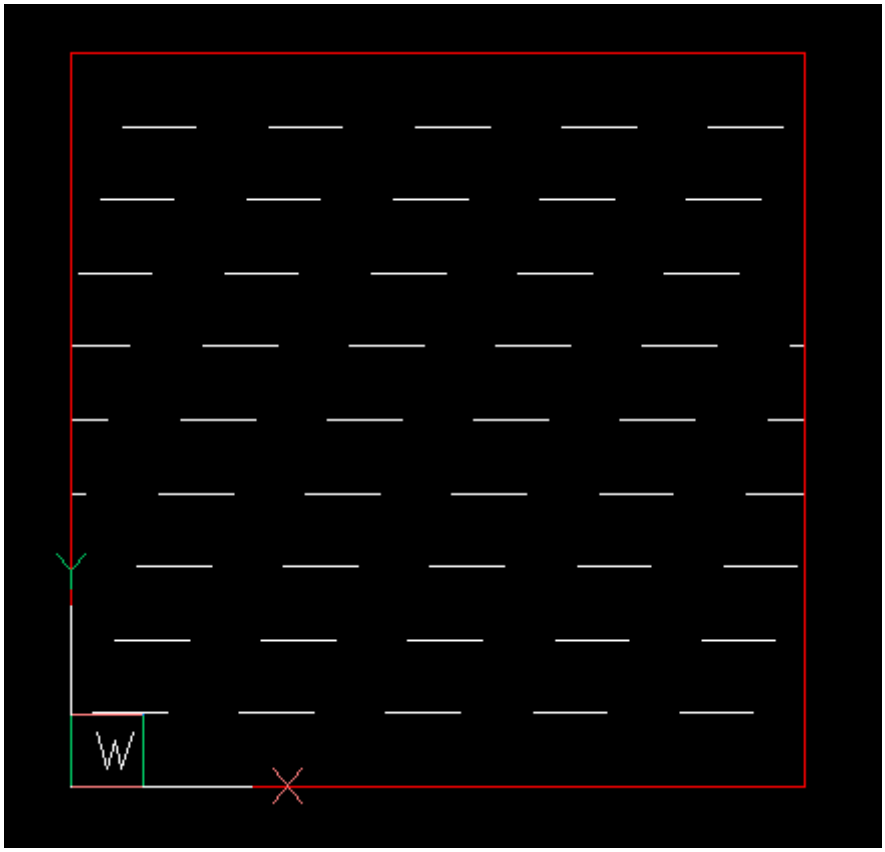
The first example creates a simple pattern of horizontal solid lines with a vertical distance of 0.5 drawing units.

```
import ezdxf

doc = ezdxf.new("R2010")
msp = doc.modelspace()
hatch = msp.add_hatch()
hatch.set_pattern_fill(
    "MyPattern",
    color=7,
    angle=0,
    scale=1.0,
    style=0,  # normal hatching style
    pattern_type=0,  # user-defined
    # pattern definition as list of:
    # [angle in degree, origin as 2d vector, offset as 2d vector, line pattern]
    # line pattern is a solid line
    definition=[[0, (0, 0), (0, 0.5), []]],

)
points = [(0, 0), (10, 0), (10, 10), (0, 10)]
hatch.paths.add_polyline_path(points)
msp.add_lwpolyline(points, close=True, dxfattribs={"color": 1})
doc.saveas("user_defined_hatch_pattern.dxf")
```



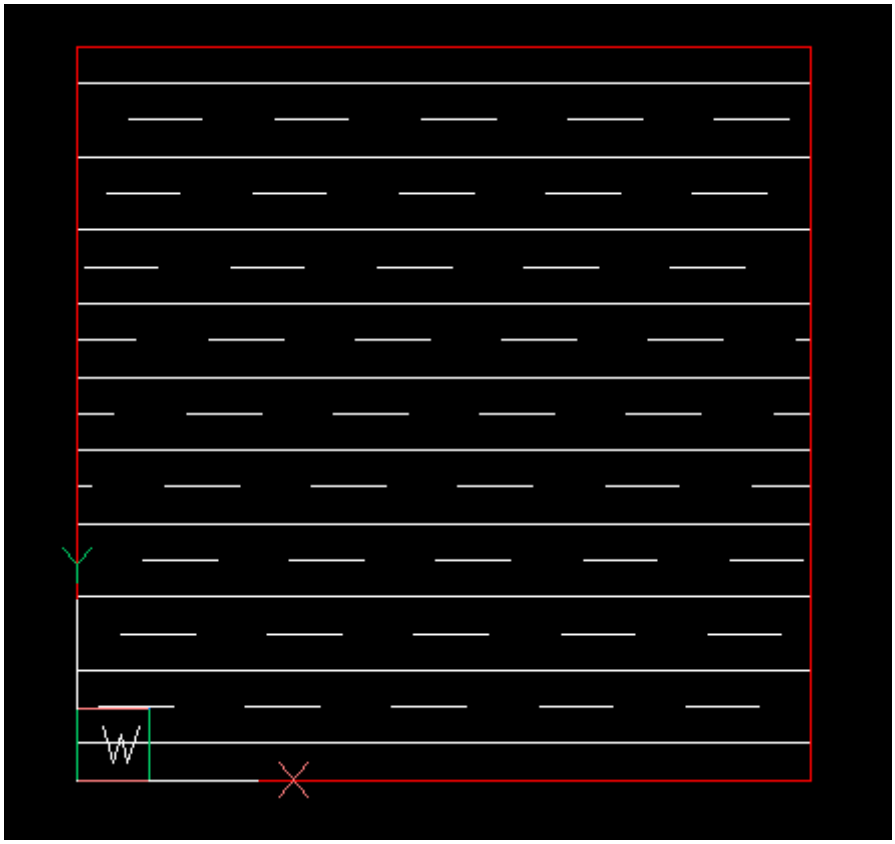
The next example shows how the *offset* value works:

```
# -x-x-x- snip -x-x-x-
hatch = msp.add_hatch()
hatch.set_pattern_fill(
    "MyPattern",
    color=7,
    angle=0,
    scale=1.0,
    style=0,  # normal hatching style
    pattern_type=0,  # user-defined
    # the line pattern is a dashed line:  - - - -
    # the offset is 1 unit vertical and 0.3 units horizontal
    # [angle in degree, origin as 2d vector, offset as 2d vector, line pattern]
    definition=[[0, (0, 0), (0.3, 1), [1, -1]]],

)
# -x-x-x- snip -x-x-x-
```



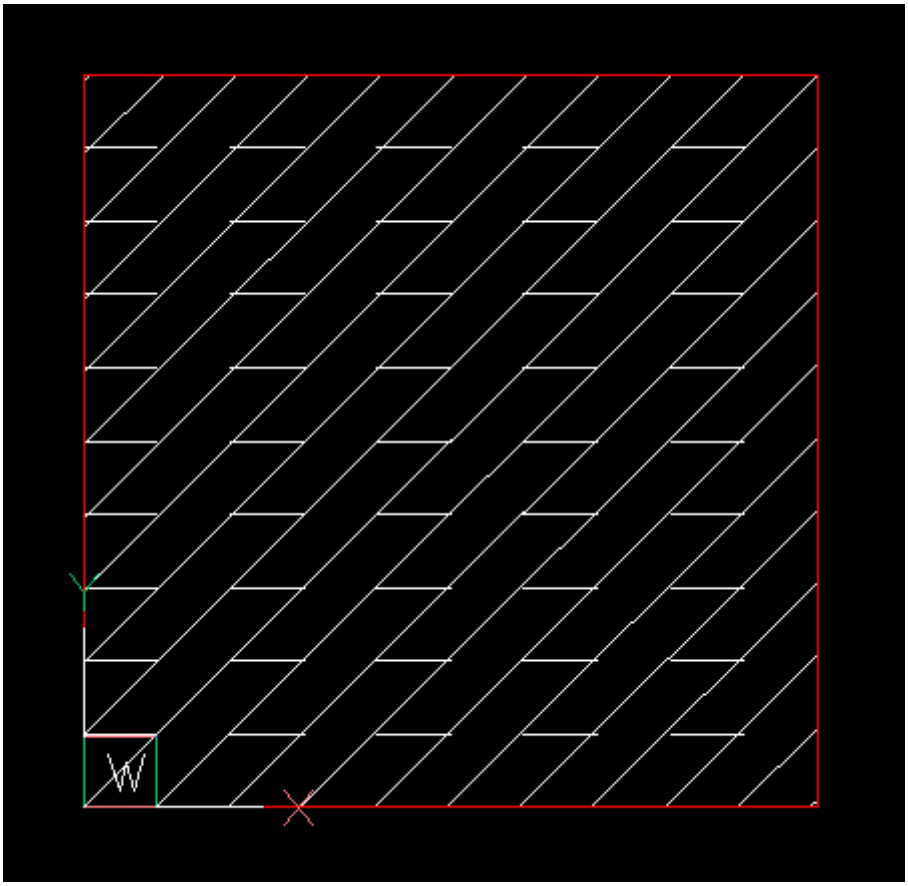
The next example combines two parallel hatch lines, the origin defines how the hatch lines are offset from each other:

```
# -x-x-x- snip -x-x-x-
hatch = msp.add_hatch()
hatch.set_pattern_fill(
    "MyPattern",
    color=7,
    angle=0,
    scale=1.0,
    style=0,  # normal hatching style
    pattern_type=0,  # user-defined
    # [angle in degree, origin as 2d vector, offset as 2d vector, line pattern]
    definition=[
        [0, (0, 0), (0.3, 1), [1, -1]],  # dashed line
        [0, (0, 0.5), (0, 1), []],  # solid line
    ],
)
# -x-x-x- snip -x-x-x-
```



The next example combines two hatch lines with different angles. The origins can be the same for this example. The `Vec2` class is used to calculate the offset value for a normal distance of 0.7 drawing units between the slanted lines:

```
from ezdxf.math import Vec2

# -x-x-x- snip -x-x-x-
hatch = msp.add_hatch()
# offset vector for a normal distance of 0.7 for a 45 deg slanted hatch line
offset = Vec2.from_deg_angle(45 + 90, length=0.7)
hatch.set_pattern_fill(
    "MyPattern",
    color=7,
    angle=0,
    scale=1.0,
    style=0,  # normal hatching style
    pattern_type=0,  # user-defined
    # [angle in degree, origin as 2d vector, offset as 2d vector, line pattern]
    definition=[
        [0, (0, 0), (0, 1), [1, -1]],  # horizontal dashed line
        [45, (0, 0), offset, []],  # slanted solid line
    ],
)
# -x-x-x- snip -x-x-x-
```



6.5.17 Tutorial for Image and ImageDef

This example shows how to use a raster image in a DXF document. Each IMAGE entity requires an associated IMAGEDEF entity in the objects section, which stores the filename of the linked image and the size in pixels. Multiple IMAGE entities can share the same IMAGEDEF entity.

Important: The raster image is NOT embedded in the DXF file!

```
import ezdxf

# The IMAGE entity requires the DXF R2000 format or later.
doc = ezdxf.new("R2000")

# The IMAGEDEF entity is like a block definition, it just defines the image.
my_image_def = doc.add_image_def(
    filename="mycat.jpg", size_in_pixel=(640, 360)
)


msp = doc.modelspace()
# The IMAGE entity is like the INSERT entity, it's just an image reference,
# and there can be multiple references to the same picture in a DXF document.

# 1st image reference
```

(continues on next page)

(continued from previous page)

```
msp.add_image(
    insert=(2, 1),
    size_in_units=(6.4, 3.6),
    image_def=my_image_def,
    rotation=0
)
# 2nd image reference
msp.add_image(
    insert=(4, 5),
    size_in_units=(3.2, 1.8),
    image_def=my_image_def,
    rotation=30
)

# Get existing image definitions from the OBJECTS section:
image_defs = doc.objects.query("IMAGEDEF")

doc.saveas("dxf_with_cat.dxf")
```

6.5.18 Tutorial for Underlay and UnderlayDefinition

This example shows hot to insert a a PDF, DWF, DWFx or DGN file as drawing underlay. Each UNDERLAY entity requires an associated UNDERLAYDEF entity in the objects section, which stores the filename of the linked document and the parameters of the underlay. Multiple UNDERLAY entities can share the same UNDERLAYDEF entity.

Important: The underlay file is NOT embedded into the DXF file:

```
import ezdxf


doc = ezdxf.new('AC1015')  # underlay requires the DXF R2000 format or later
my_underlay_def = doc.add_underlay_def(filename='my_underlay.pdf', name='1')
# The (PDF)DEFINITION entity is like a block definition, it just defines the underlay
# 'name' is misleading, because it defines the page/sheet to be displayed
# PDF: name is the page number to display
# DGN: name='default' ???
# DWF: ????

msp = doc.modelspace()
# add first underlay
msp.add_underlay(my_underlay_def, insert=(2, 1, 0), scale=0.05)
# The (PDF)UNDERLAY entity is like the INSERT entity, it creates an underlay␣
↪reference,
# and there can be multiple references to the same underlay in a drawing.

msp.add_underlay(my_underlay_def, insert=(4, 5, 0), scale=.5, rotation=30)

# get existing underlay definitions, Important: UNDERLAYDEFs resides in the objects␣
↪section
pdf_defs = doc.objects.query('PDFDEFINITION')  # get all pdf underlay defs in drawing

doc.saveas("dxf_with_underlay.dxf")
```

6.5.19 Tutorial for MultiLeader

A multileader object typically consists of an arrowhead, a horizontal landing (a.k.a. "dogleg"), a leader line or curve, and either a MTEXT object or a BLOCK.

Factory methods of the *BaseLayout* class to create new *MultiLeader* entities:

- *add_multileader_mtext()*
- *add_multileader_block()*

Because of the complexity of the MULTILEADER entity, the factory method *add_multileader_mtext()* returns a *MultiLeaderMTextBuilder* instance to build a new entity and the factory method *add_multileader_block()* returns a *MultiLeaderBlockBuilder* instance.

Due of the lack of good documentation it's not possible to support all combinations of MULTILEADER properties with decent quality, so stick to recipes and hints shown in this tutorial to get usable results otherwise, you will enter uncharted territory.

The rendering result of the MULTILEADER entity is highly dependent on the CAD application. The MULTILEADER entity does not have a pre-rendered anonymous block of DXF primitives like all DIMENSION entities, so results may vary from CAD application to CAD application. The general support for this entity is only good in Autodesk products other CAD applications often struggle when rendering MULTILEADERS, even my preferred testing application BricsCAD has rendering issues.

Important: MULTILEADER support has flaws in many CAD applications except Autodesk products!

See also:

- *ezdxf.render.MultiLeaderBuilder* classes
- *ezdxf.entities.MultiLeader* class
- *ezdxf.entities.MLeaderStyle* class
- *ezdxf.tools.text.MTextEditor* class
- *MULTILEADER Internals*

MTEXT Quick Draw

Full Python script: mtext_quick_leader.py

The `quick_leader()` method of a MTEXT - MULTILEADER entity constructs the geometry parameters in reverse manner, starting from a given target point:

DXF document setup:

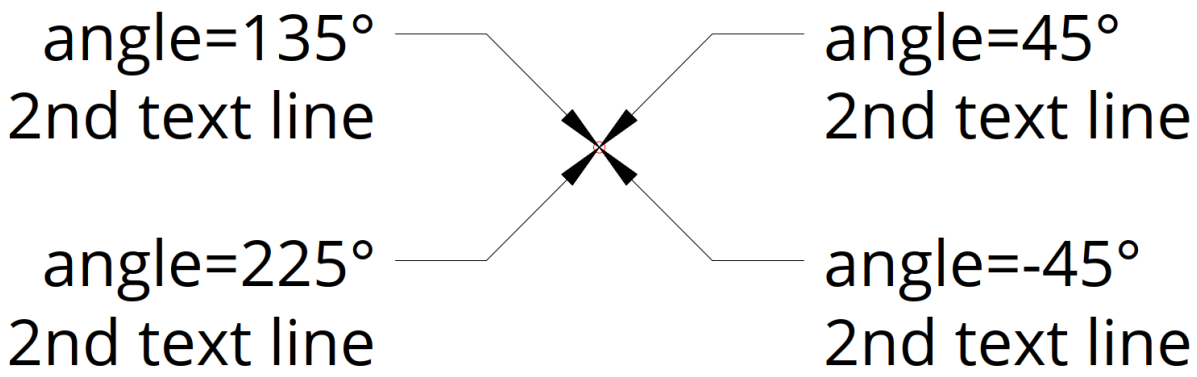
```
    doc = ezdxf.new(setup=True)
    # Create a new custom MLEADERSTYLE:
    mleaderstyle = doc.mleader_styles.duplicate_entry("Standard", "EZDXF")
    # The required TEXT style "OpenSans" was created by ezdxf.new() because setup is␣
↪True:
    mleaderstyle.set_mtext_style("OpenSans")
    msp = doc.modelspace()
```

Draw a red circle to mark the target point:

```
target_point = Vec2(40, 15)
msp.add_circle(
    target_point, radius=0.5, dxfattribs=GfxAttribs(color=colors.RED)
)
```

Create four horizontal placed MULTILEADER entities pointing at the target point, the first segment of the leader line is determined by an angle in this example pointing away from the target point:

```
for angle in [45, 135, 225, -45]:
    ml_builder = msp.add_multileader_mtext("EZDXF")
    ml_builder.quick_leader(
        f"angle={angle}°\n2nd text line",
        target=target_point,
        segment1=Vec2.from_deg_angle(angle, 14),
    )
```



The content is automatically aligned to the end of the leader line. The first segment is a relative vector to the target point and the optional second segment vector is relative to the end of the first segment. The default connection type is horizontal but can be changed to vertical:

A smaller text size is required:

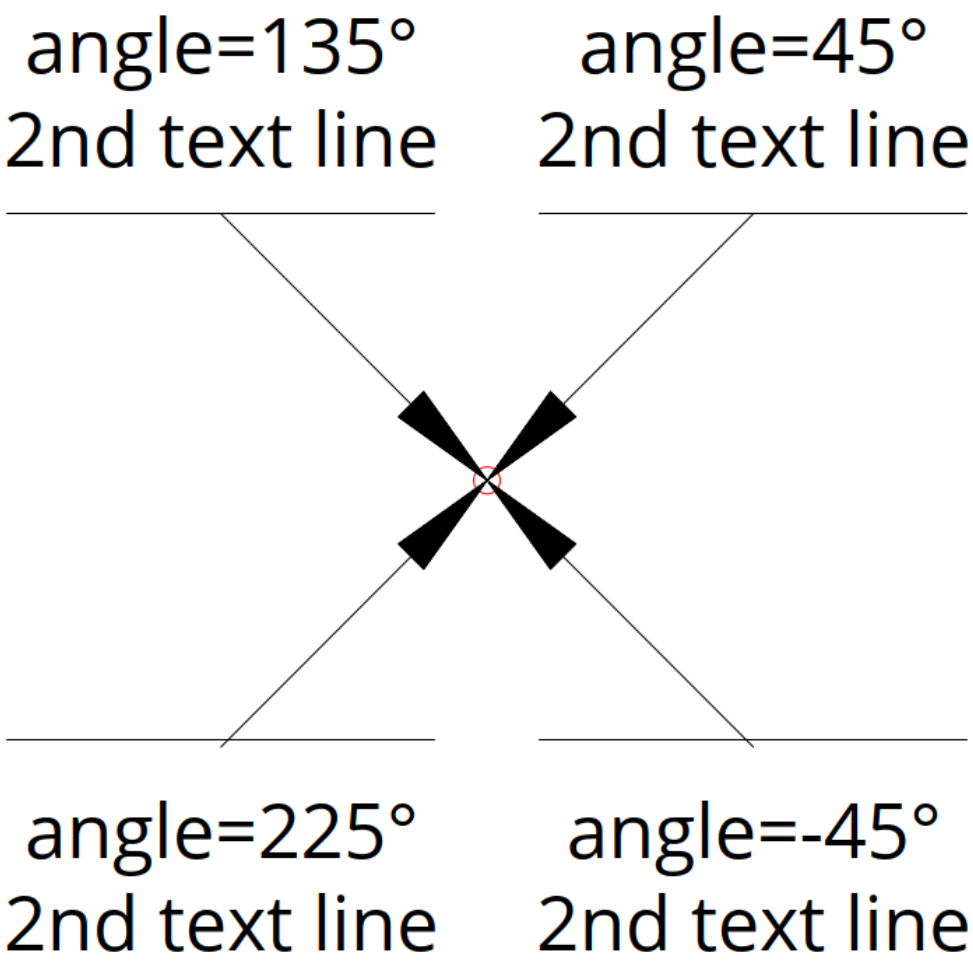
```
mleaderstyle = doc.mleader_styles.duplicate_entry("Standard", "EZDXF")
mleaderstyle.set_mtext_style("OpenSans")
mleaderstyle.dxf.char_height = 2.0  # set the default char height of MTEXT
```

Adding vertical placed MULTILEADER entities:

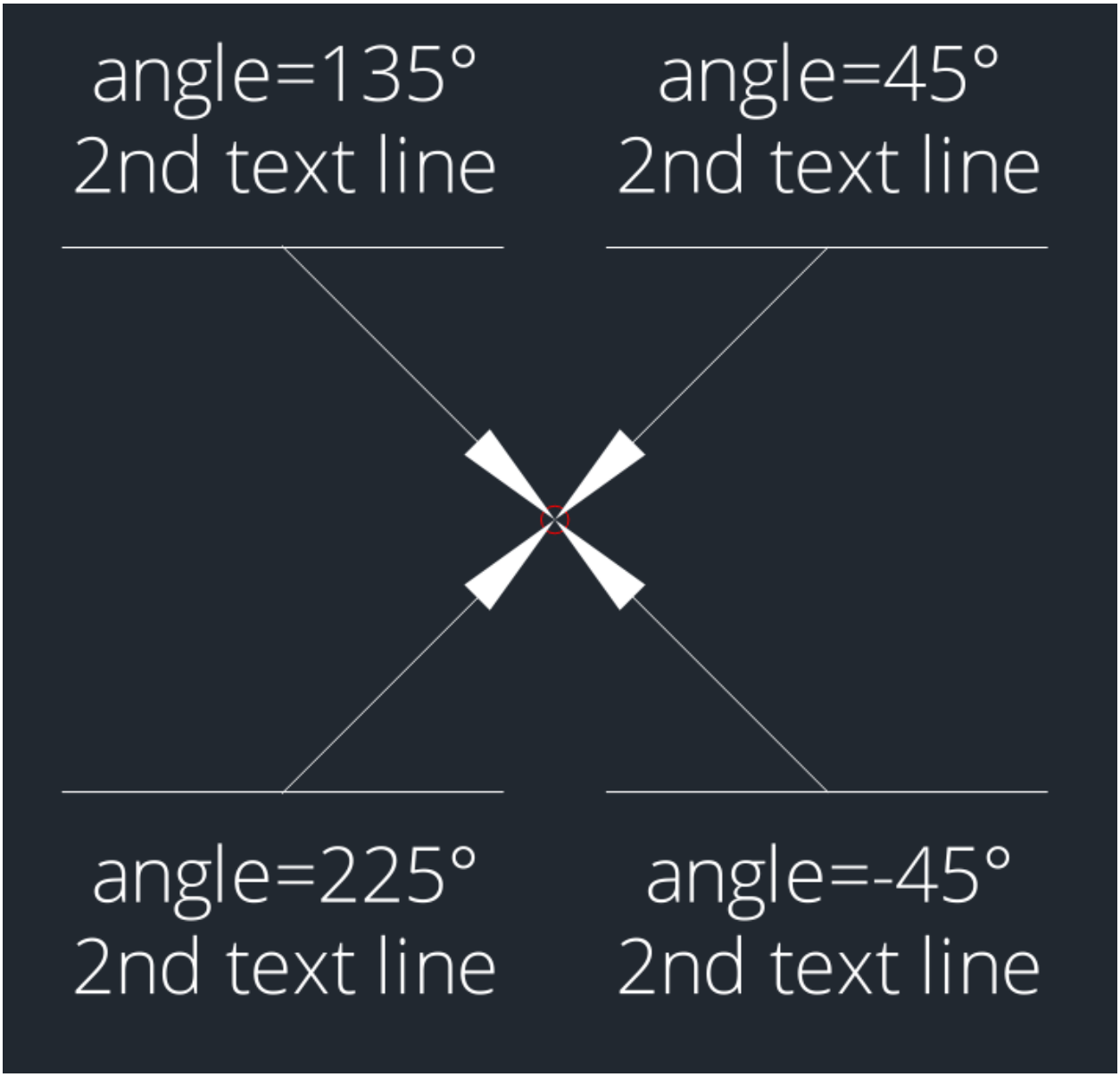
```
for angle in [45, 135, 225, -45]:
    ml_builder = msp.add_multileader_mtext("EZDXF")
    ml_builder.quick_leader(
        f"angle={angle}°\n2nd text line",
        target=target_point,
        segment1=Vec2.from_deg_angle(angle, 14),
        connection_type=mleader.VerticalConnection.center_overline,
```

This example already shows the limitation caused by different text renderings in various CAD applications. The *ezdxf* text measurement by *matplotlib* is different to AutoCAD and BricsCAD and the result is a misalignment of the overline and the leader line.

The DXF file shown in BricsCAD:



The same DXF file shown with the `ezdxf view` command (drawing add-on):



My advice is to avoid vertical placed MULTILEADER entities at all and for horizontal placed MULTILEADER entities avoid styles including an "underline" or an "overline".

The `quick_leader()` method is not very customizable for ease of use, but follows the settings of the associated *MLeaderStyle*.

The following sections show how to have more control when adding MULTILEADER entities.

Create MTEXT Content

Full Python script: mtext_content.py

This section shows how to create a MULTILEADER entity with MTEXT content the manual way with full control over all settings.

For good results the MTEXT alignment should match the leader connection side, e.g. if you attach leaders to the left side also align the MTEXT to the left side, for leaders attached at the right side, align the MTEXT to the right side and if you attach leaders at both sides one side will fit better than the other or maybe a center aligned MTEXT is a good solution, for further details see section *MTEXT Alignment*.

The first example uses the default connection type of the MLEADERSTYLE "Standard" which is "middle of the top line" for left and right attached leaders. The render UCS for this example is the WCS to keep things simple.

Create a new MULTILEADER entity.

```
    msp = doc.modelspace()
```

Set MTEXT content, text style and alignment.

```
    ml_builder = msp.add_multileader_mtext("Standard")
    ml_builder.set_content(
        "Line1\nLine2",
        style="OpenSans",
        alignment=mleader.TextAlignment.left,  # set MTEXT alignment!
```

Add the first leader on the left side. The leader points always to the first given vertex and all vertices are given in render UCS coordinates (= WCS in this example).

```
    )
```

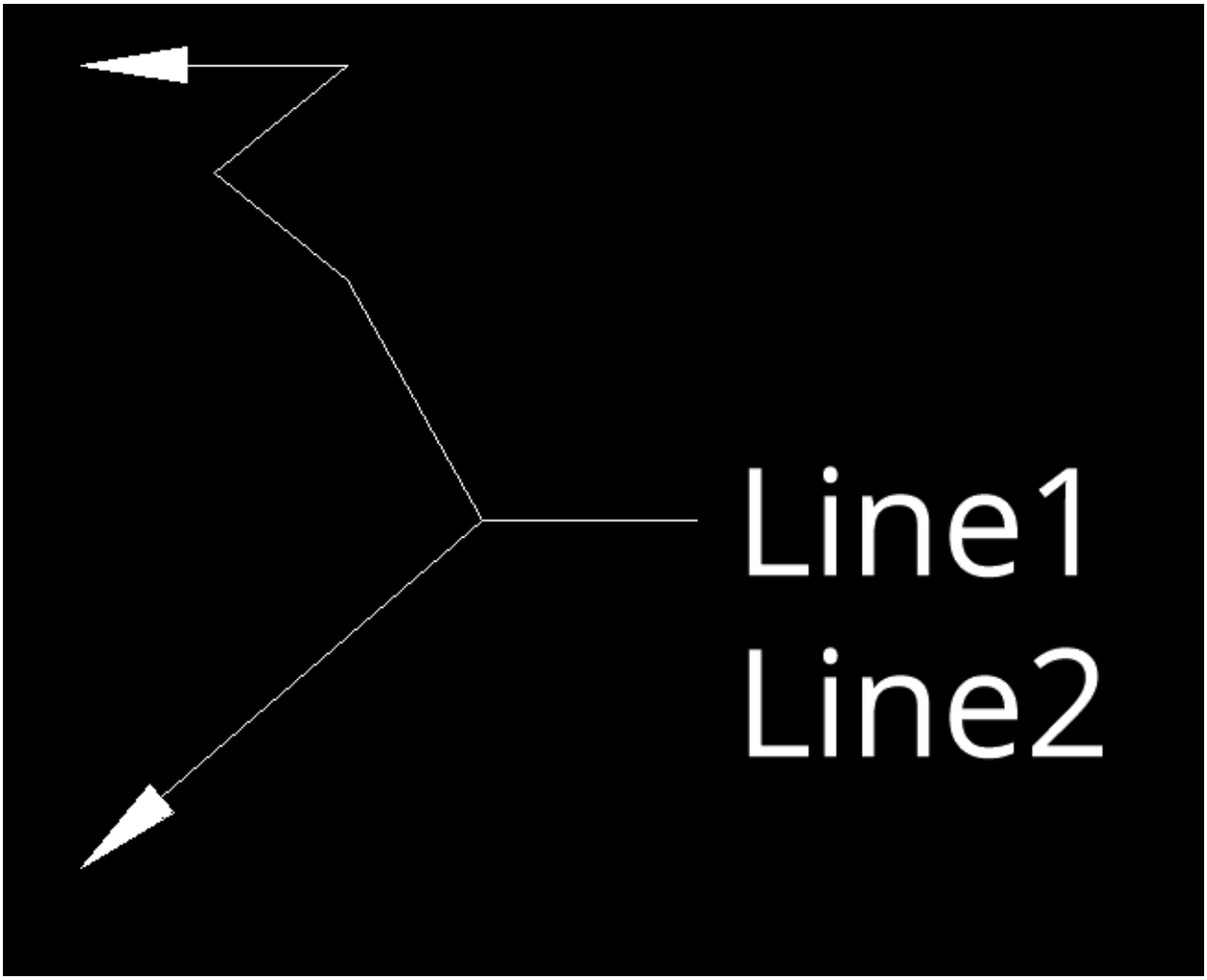
More than one vertex per leader can be used:

```
    ml_builder.add_leader_line(mleader.ConnectionSide.left, [Vec2(-20, -15)])
    ml_builder.add_leader_line(
        mleader.ConnectionSide.left,
        [Vec2(-20, 15), Vec2(-10, 15), Vec2(-15, 11), Vec2(-10, 7)],
```

The insert point of the `build()` method is the alignment point for the MTEXT content.

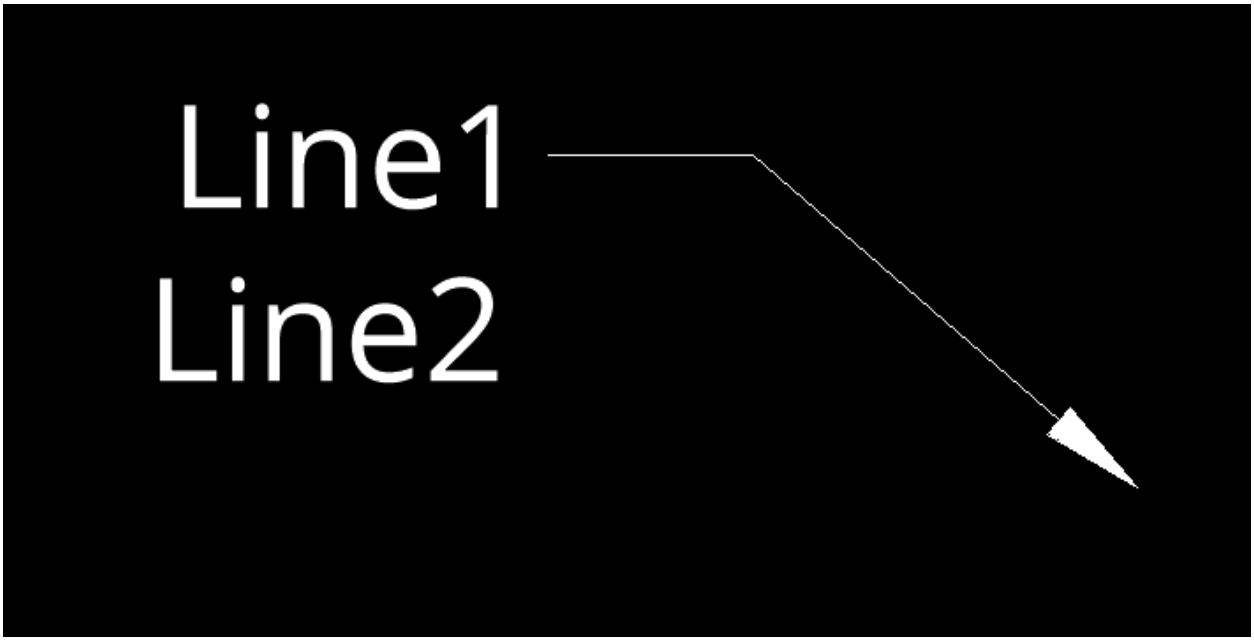
```
    )
```

The "dogleg" settings are defined by the MLEADERSTYLE "Standard".



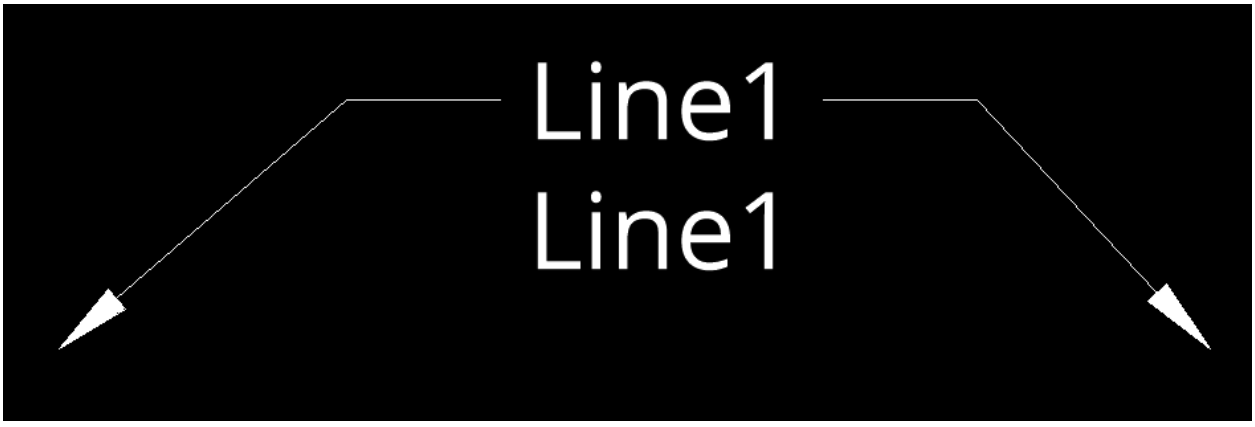
This example shows a leader attached to the right side and the MTEXT aligned to the right side.

```
msp = doc.modelspace()
ml_builder = msp.add_multileader_mtext("Standard")
ml_builder.set_content(
    "Line1\nLine2",
    style="OpenSans",
    alignment=mleader.TextAlignment.right,  # set MTEXT alignment!
)
ml_builder.add_leader_line(mleader.ConnectionSide.right, [Vec2(40, -15)])
```



This example shows two leaders attached to both sides and the MTEXT aligned to the left side, which shows that the right landing gap (space between text and start of vertex) is bigger than the gap on the left size. This is due to the different text size calculations from AutoCAD/BricsCAD and Matplotlib. The longer the text, the greater the error.

```
msp = doc.modelspace()
ml_builder = msp.add_multileader_mtext("Standard")
ml_builder.set_content(
    "Line1\nLine1",
    style="OpenSans",
    alignment=mleader.TextAlignment.left,  # set MTEXT alignment!
)
ml_builder.add_leader_line(mleader.ConnectionSide.left, [Vec2(-20, -15)])
ml_builder.add_leader_line(mleader.ConnectionSide.right, [Vec2(40, -15)])
```



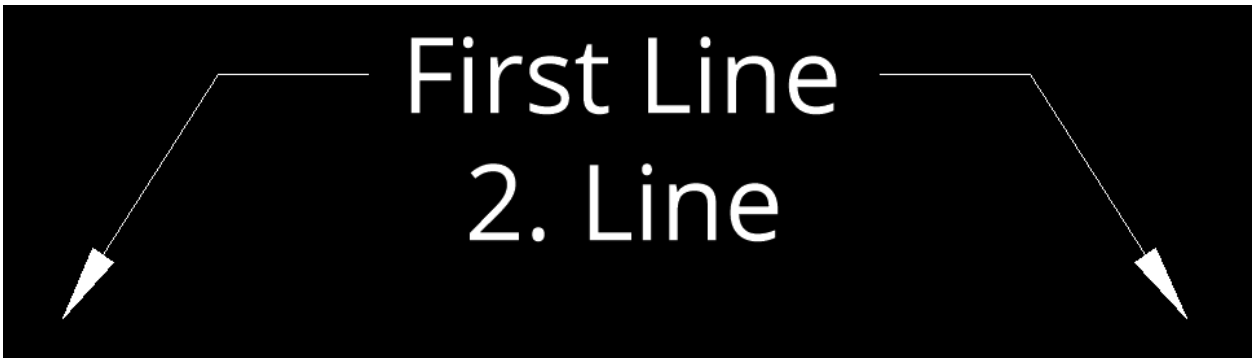
A centered MTEXT alignment gives a more even result.

```
msp = doc.modelspace()
ml_builder = msp.add_multileader_mtext("Standard")
ml_builder.set_content(
    "First Line\n2. Line",
    style="OpenSans",
```

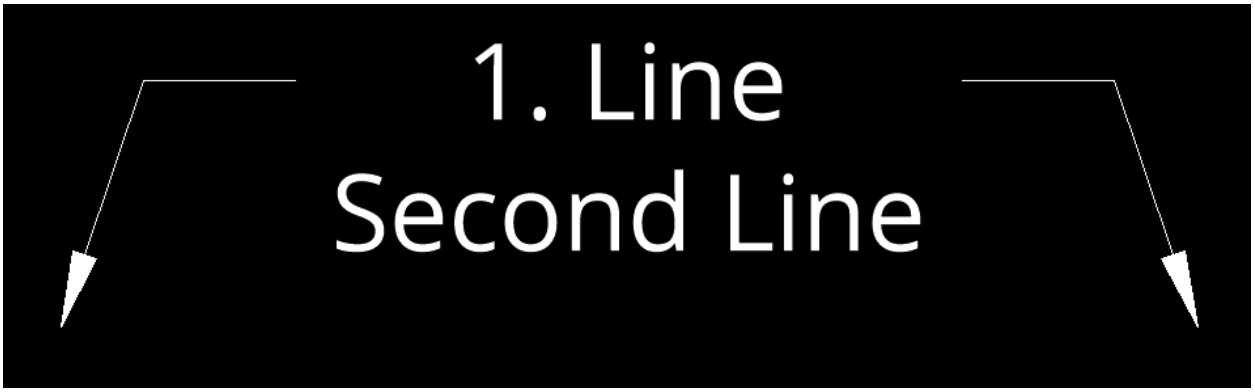
(continues on next page)

(continued from previous page)

```
        alignment=mleader.TextAlignment.center,  # set MTEXT alignment!
    )
    ml_builder.add_leader_line(mleader.ConnectionSide.left, [Vec2(-20, -15)])
    ml_builder.add_leader_line(mleader.ConnectionSide.right, [Vec2(40, -15)])
```



But even this has its disadvantages, the attachment calculation is always based on the bounding box of the MTEXT content.



MTEXT Connection Types

There are four connection sides defined by the enum *ezdxf.render.ConnectionSide*:

- left
- right
- top
- bottom

The MultiLeader entity supports as the name says multiple leader lines, but all have to have a horizontal (left/right) connection side or a vertical (top/bottom) connection side, it's not possible to mix left/right and top/bottom connection sides. This is determined by the DXF format.

There are different connection types available for the horizontal and the vertical connection sides. All leaders connecting to the same side have the same connection type. The horizontal connection sides support following connection types, defined by the enum *ezdxf.render.HorizontalConnection*:

- by_style
- top_of_top_line
- middle_of_top_line

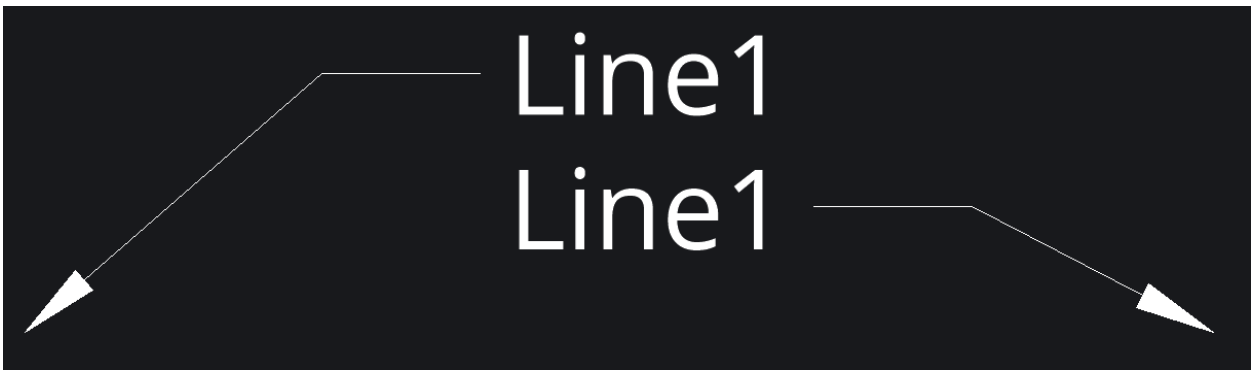
- middle_of_text
- middle_of_bottom_line
- bottom_of_bottom_line
- bottom_of_bottom_line_underline (not recommended)
- bottom_of_top_line_underline (not recommended)
- bottom_of_top_line
- bottom_of_top_line_underline_all (not recommended)

The vertical connection sides support following connection types, defined by the enum *ezdxf.render.VerticalConnection*:

- by_style
- center
- center_overline (not recommended)

The connection type for each side can be set by the method *set_connection_types()*, the default for all sides is *by_style*:

```
    ml_builder.add_leader_line(mleader.ConnectionSide.right, [Vec2(40, -15)])
    ml_builder.set_connection_types(
        left=mleader.HorizontalConnection.middle_of_top_line,
        right=mleader.HorizontalConnection.middle_of_bottom_line,
```



Hint: As shown in the quick draw section using connection types including underlines or overlines do not render well in AutoCAD/BricsCAD because of the different text measurement of *matplotlib*, therefore it's not recommended to use any of these connection types when creating MULTILEADERS by *ezdxf*.

MTEXT Alignment

In contrast to the standalone MTEXT entity supports the MTEXT content entity only three text alignments defined by the enum *ezdxf.render.TextAlignment*.

- left
- center
- right

The MTEXT alignment is set as argument *alignment* of the `set_content()` method and the alignment point is the insert point of the *`build()`* method.

Create BLOCK Content

Full Python script: block_content.py

This section shows how to create a MULTILEADER entity with BLOCK content the manual way with full control over all settings.

The BLOCK content consist of a BLOCK layout and optional ATTDEF entities which defines the location and DXF attributes of dynamically created ATTRIB entities.

Create the BLOCK content, the full `create_square_block()` function can be found in the block_content.py script.

```
    msp = doc.modelspace()
    block = create_square_block(
        doc, size=8.0, margin=0.25, base_point=base_point
```

Create the MULTILEADER and set the content:

```
    )
    ml_builder = msp.add_multileader_block(style="Standard")
    ml_builder.set_content(
        name=block.name, alignment=mleader.BlockAlignment.insertion_point
```

Set the BLOCK attribute content as text:

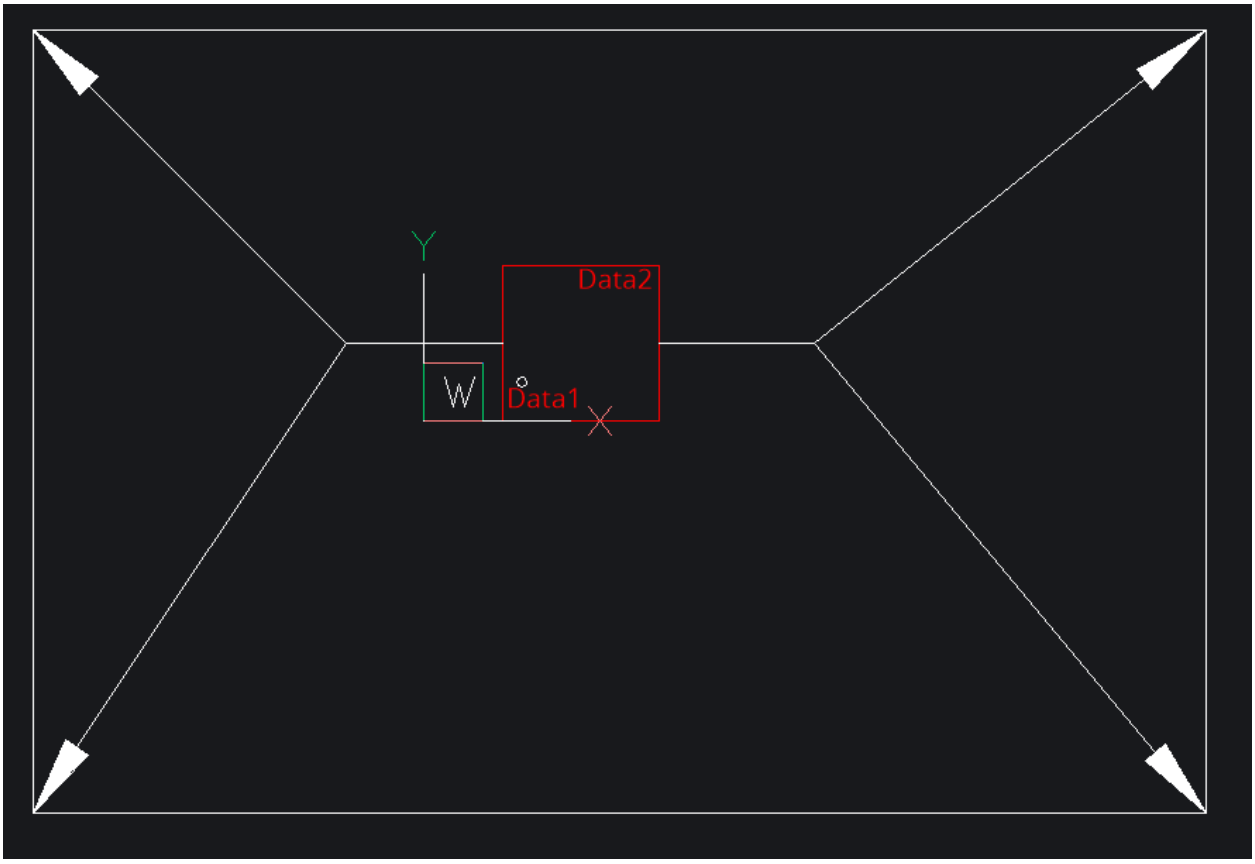
```
    )
    ml_builder.set_attribute("ONE", "Data1")
```

Add some leader lines to the left and right side of the BLOCK:

Construction plane of the entity is defined by a render UCS. The leader lines vertices are expected in render UCS coordinates, which means relative to the UCS origin and this example shows the simple case where the UCS is the WCS which is also the default setting.

```


    ml_builder.add_leader_line(mleader.ConnectionSide.right, [Vec2(x2, y1)])
    ml_builder.add_leader_line(mleader.ConnectionSide.right, [Vec2(x2, y2)])
    ml_builder.add_leader_line(mleader.ConnectionSide.left, [Vec2(x1, y1)])
```

Last step is to build the final MULTILEADER entity. This example uses the alignment type *insertion_point* where the insert point of the `build()` method is the base point of the BLOCK:



The result is shown in BricsCAD as expected, although BricsCAD shows “Center extents” as attachment type in the properties dialog instead of the correct attachment type “Insertion point”.

BLOCK Connection Types

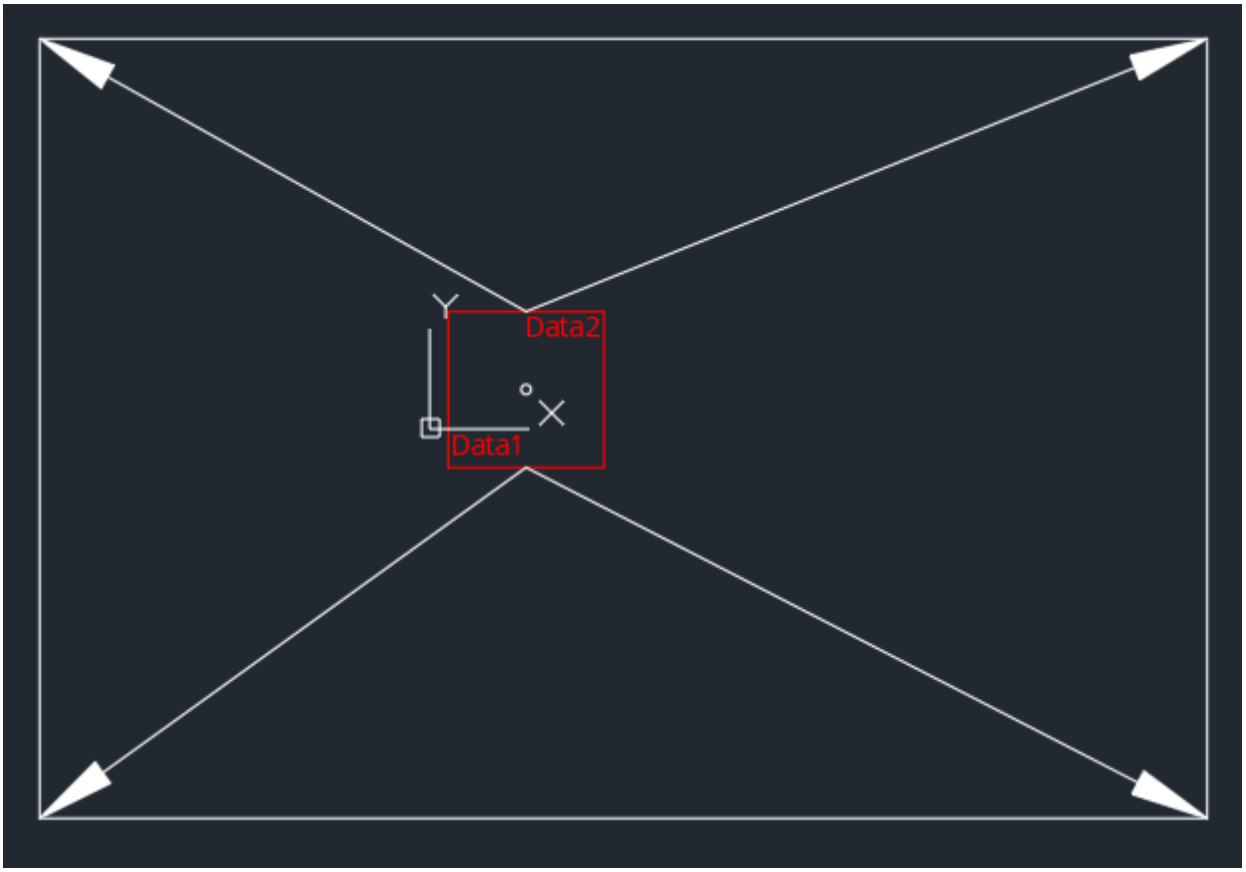
There are four connection sides defined by the enum *ezdxf.render.ConnectionSide*:

- left
- right
- top
- bottom

The connection point for leader lines is always the center of the side of the block bounding box the leader is connected to and has the same limitation as for the MTEXT content, it’s not possible to mix the connection sides left/right and top/bottom.

The connection side is set when adding the leader line by the *add_leader_line()* method.

Unfortunately BricsCAD has an error in version 22.2.03 and renders all connection types as left/right, this is top/bottom connection shown in Autodesk TrueView 2022:



The top/bottom connection type does not support the “dogleg” feature.

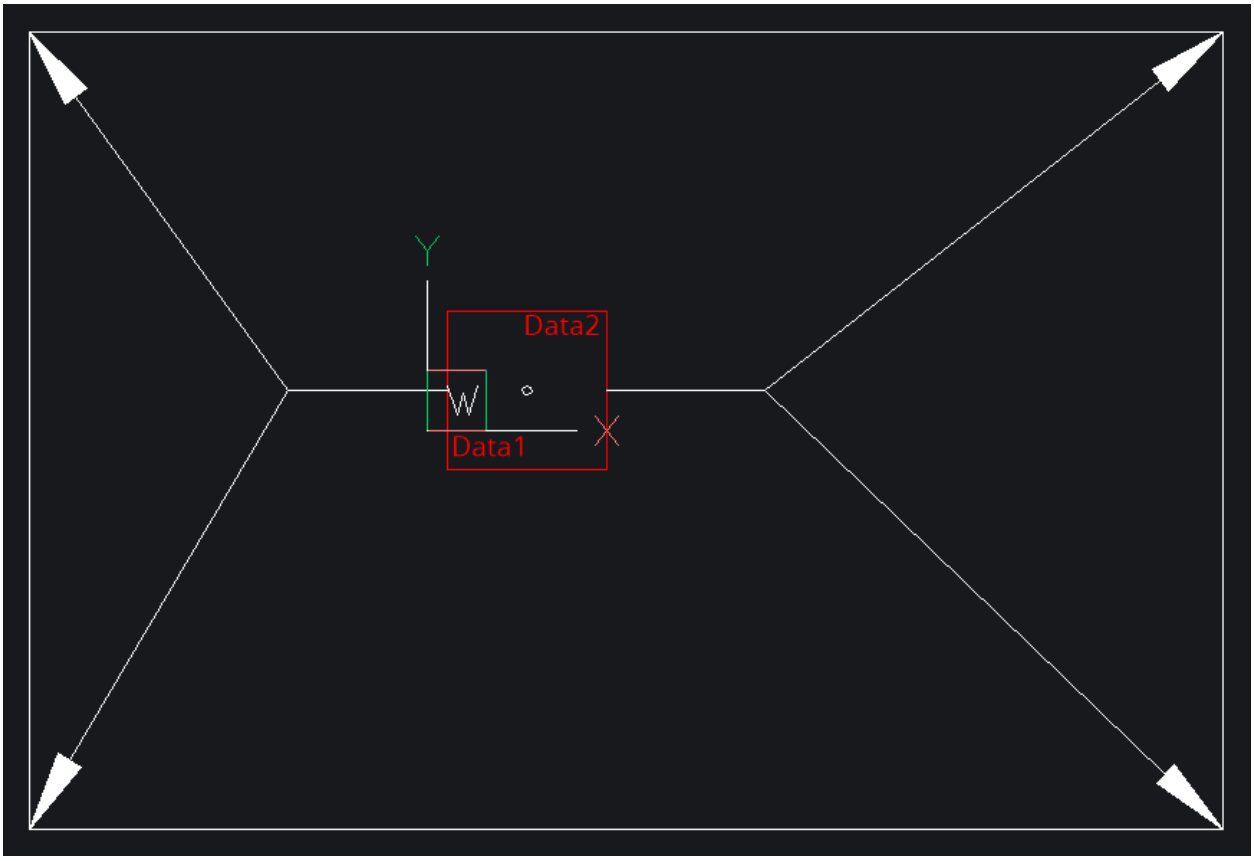
BLOCK Alignment

There are two alignments types, defined by the enum *ezdxf.render.BlockAlignment*

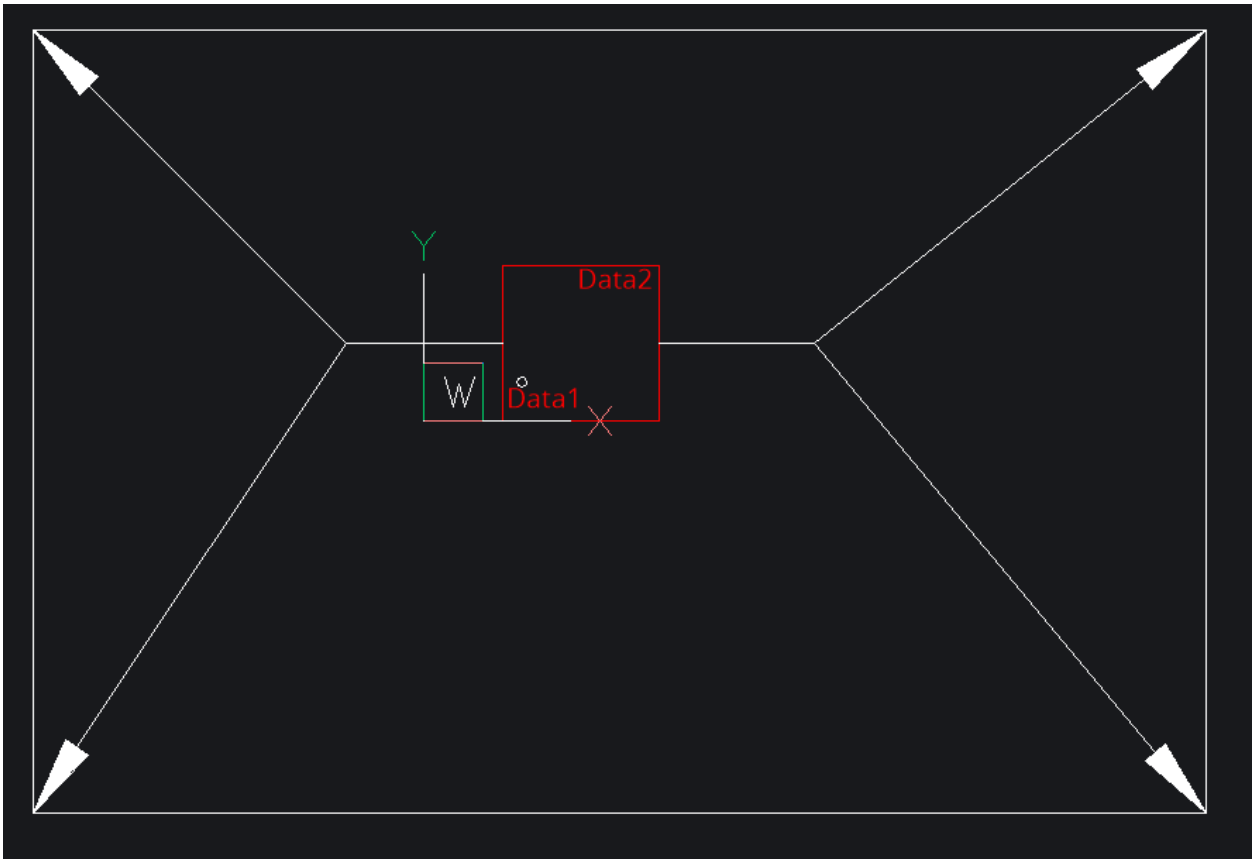
- center_extents
- insertion_point

The alignment is set by the *set_content()* method.

The alignment type *center_extent* inserts the BLOCK with the center of the bounding box at the insert point of the *build()* method. The insert point is (5, 2) in this example:



The same MULTILEADER with alignment type *insert_point*:



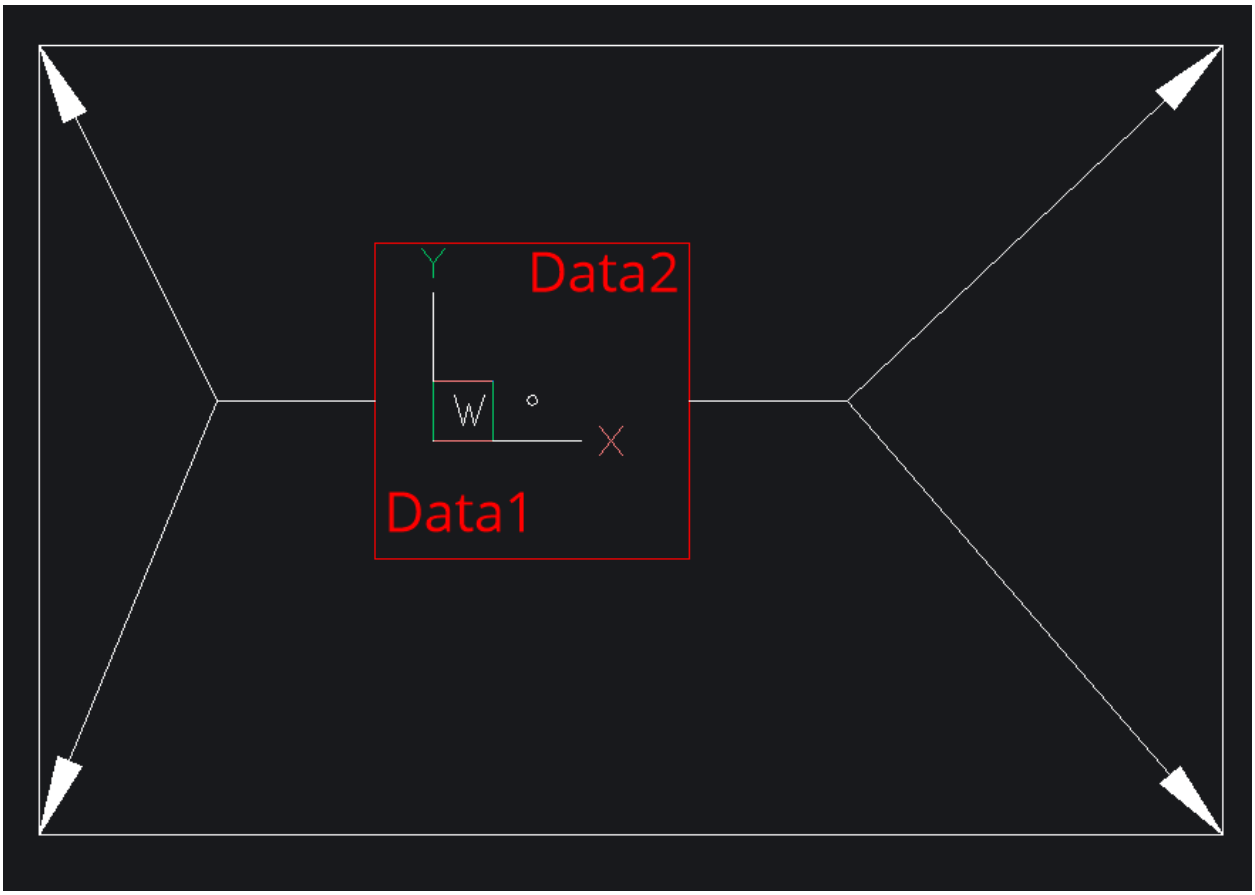
BLOCK Scaling

The BLOCK content can be scaled independently from the overall scaling of the MULTILEADER entity:

The block scaling factor is set by the *set_content()* method:

```
ml_builder.set_content(
    name=block.name, scale=2.0, alignment=mleader.BlockAlignment.center_extents
)
```

This is the first example with a block scaling factor of 2. The BLOCK and the attached ATTRIB entities are scaled but not the arrows.

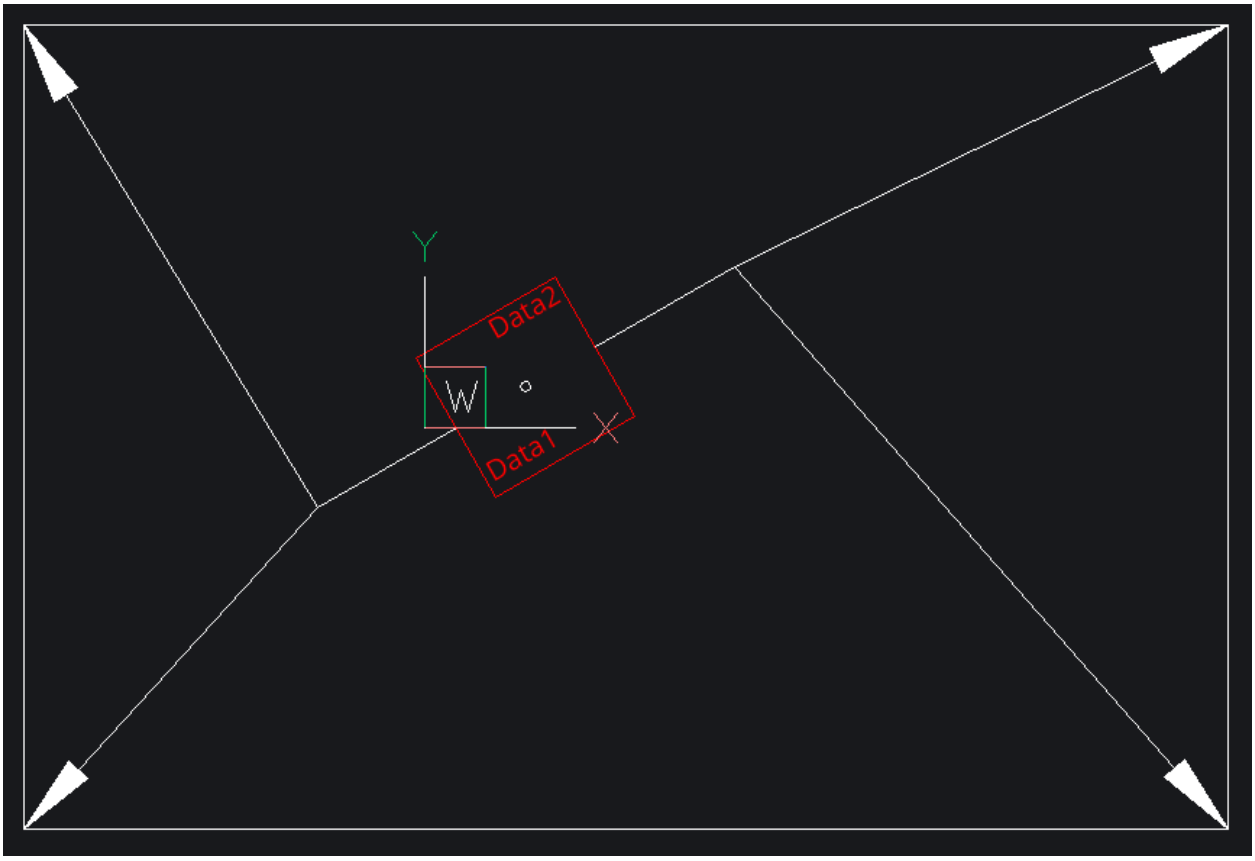


BLOCK Rotation

The rotation around the render UCS z-axis in degrees is applied by the *build()* method:

```
ml_builder.build(insert=Vec2(5, 2), rotation=30)
```

This is the first example with a rotation of 30 degrees. The BLOCK, the attached ATTRIB entities and the last connection lines (“dogleg”) are rotated.



BLOCK Attributes

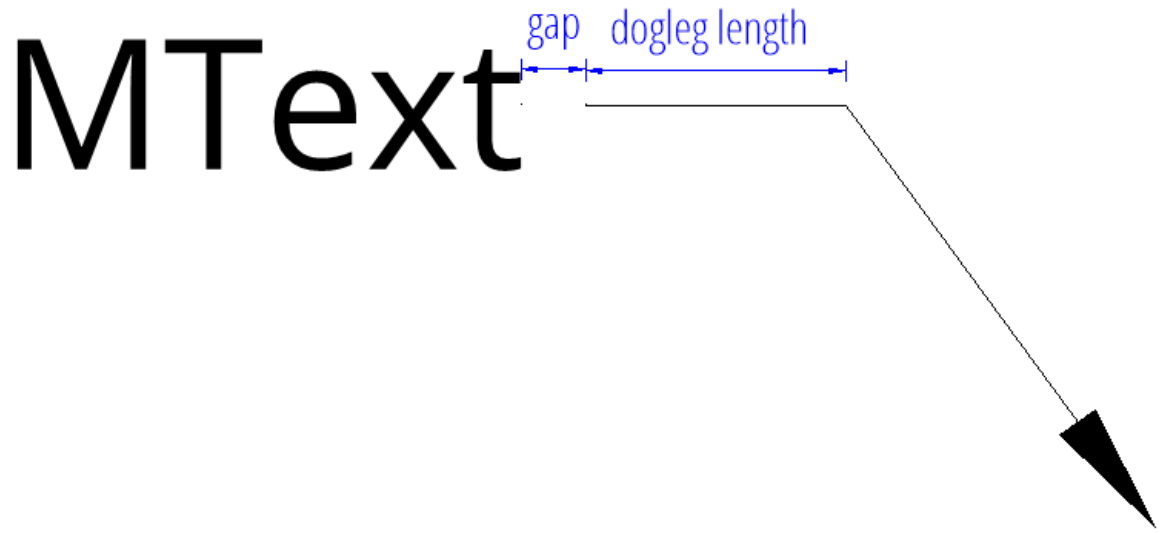
BLOCK attributes are defined as ATTDEF entities in the BLOCK layout. This ATTDEF entities will be replaced by ATTRIB entities at the rendering process of the CAD application. Only the text content and the text width factor can be changed for each MULTILEADER entity individually by the *set_attribute()* method. The ATTDEF is addressed by it's DXF *tag* attribute:

```
ml_builder.set_attribute("ONE", "Data1")
ml_builder.set_attribute("TWO", "Data2")
```

Leader Properties

"Dogleg" Properties

The "dogleg" is the last line segment from the last leader vertex to the MULTILEADER content for polyline leaders.

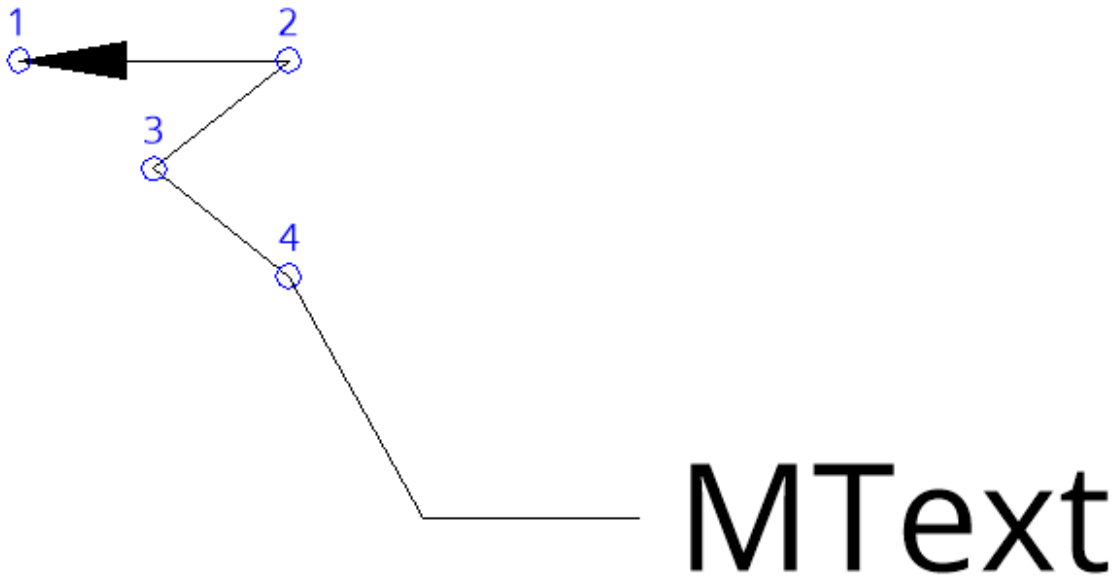


The length of the dogleg and the landing gap size is set by the *set_connection_properties()*.

Polyline Leader

A polygon leader line has only straight line segments and is added by the *add_leader_line()*:

```
ml_builder.add_leader_line(
    mleader.ConnectionSide.left,
    [Vec2(-20, 15), Vec2(-10, 15), Vec2(-15, 11), Vec2(-10, 7)],
)
```

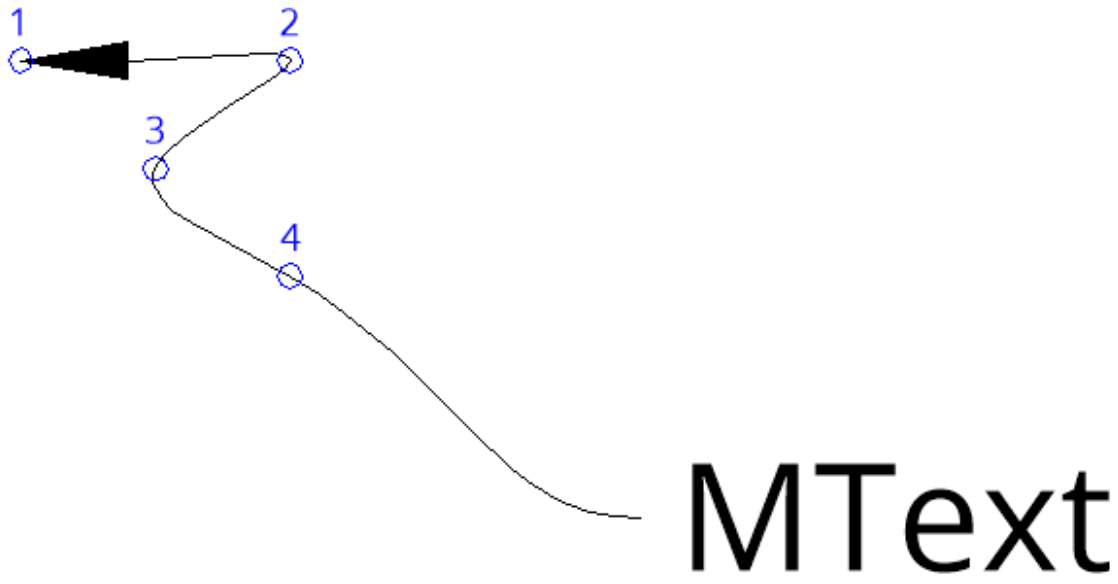


All leader line vertices have render UCS coordinates and the start- and end-vertex of the "dogleg" is calculated automatically.

Spline Leader

A spline leader line has a single curved line as leader line and is also added by the *add_leader_line()*. This is spline leader has the same vertices as the previous created polyline leader:

```
ml_builder.set_leader_properties(leader_type=mleader.LeaderType.splines)
ml_builder.add_leader_line(
    mleader.ConnectionSide.left,
    [Vec2(-20, 15), Vec2(-10, 15), Vec2(-15, 11), Vec2(-10, 7)],
)
```



The spline leader has no “dogleg” and spline leaders and polyline leaders can not be mixed in a single MULTILEADER entity.

The leader type is set by the *set_leader_properties()* method.

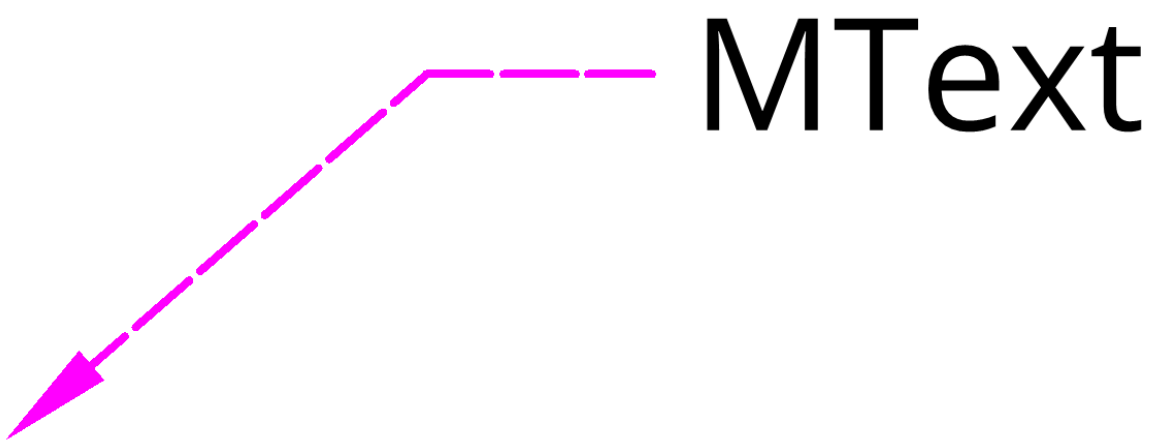
The *LeaderType* enum:

- none
- straight_lines
- splines

Line Styling

The leader color, linetype and lineweight is set by the *set_leader_properties()* method:

```
ml_builder.set_leader_properties(
    color=colors.MAGENTA,
    linetype="DASHEDX2",
    lineweight=70,
)
```

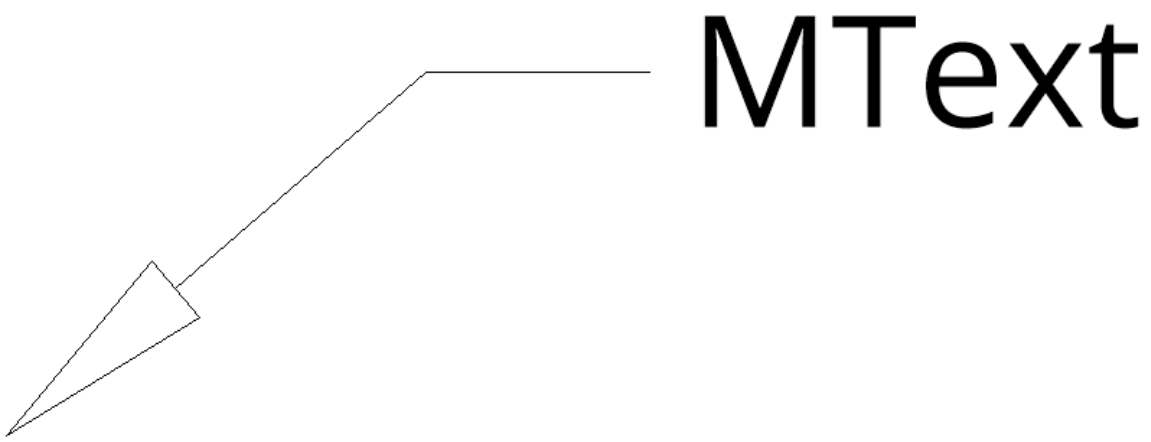


All leader lines have the same properties.

Arrowheads

The arrow head is set by the *set_arrow_properties()* method:

```
from ezdxf.render import ARROWS
ml_builder.set_arrow_properties(name=ARROWS.closed_blank, size=8.0)
```



All leader lines have the same arrow head and size. The available arrow heads are defined in the *ARROWS* object.

Overall Scaling

The overall scaling has to be applied by the *set_overall_scaling()* method and scales the MTEXT or BLOCK content **and** the arrows.

Setup MLEADERSTYLE

The *MLeaderStyle* stores many of the MULTILEADER settings but most of them are copied to the MULTILINE entity at initialization. So changing the MLEADERSTYLE style afterwards has little to no effect for existing MULTILEADER entities.

Create a new MLEADERSTYLE called "MY_STYLE" and set the MTEXT style to "OpenSans":

```
my_style = doc.mleader_styles.duplicate_entry("Standard", "MY_STYLE")
my_style.set_mtext_style("OpenSans")
```

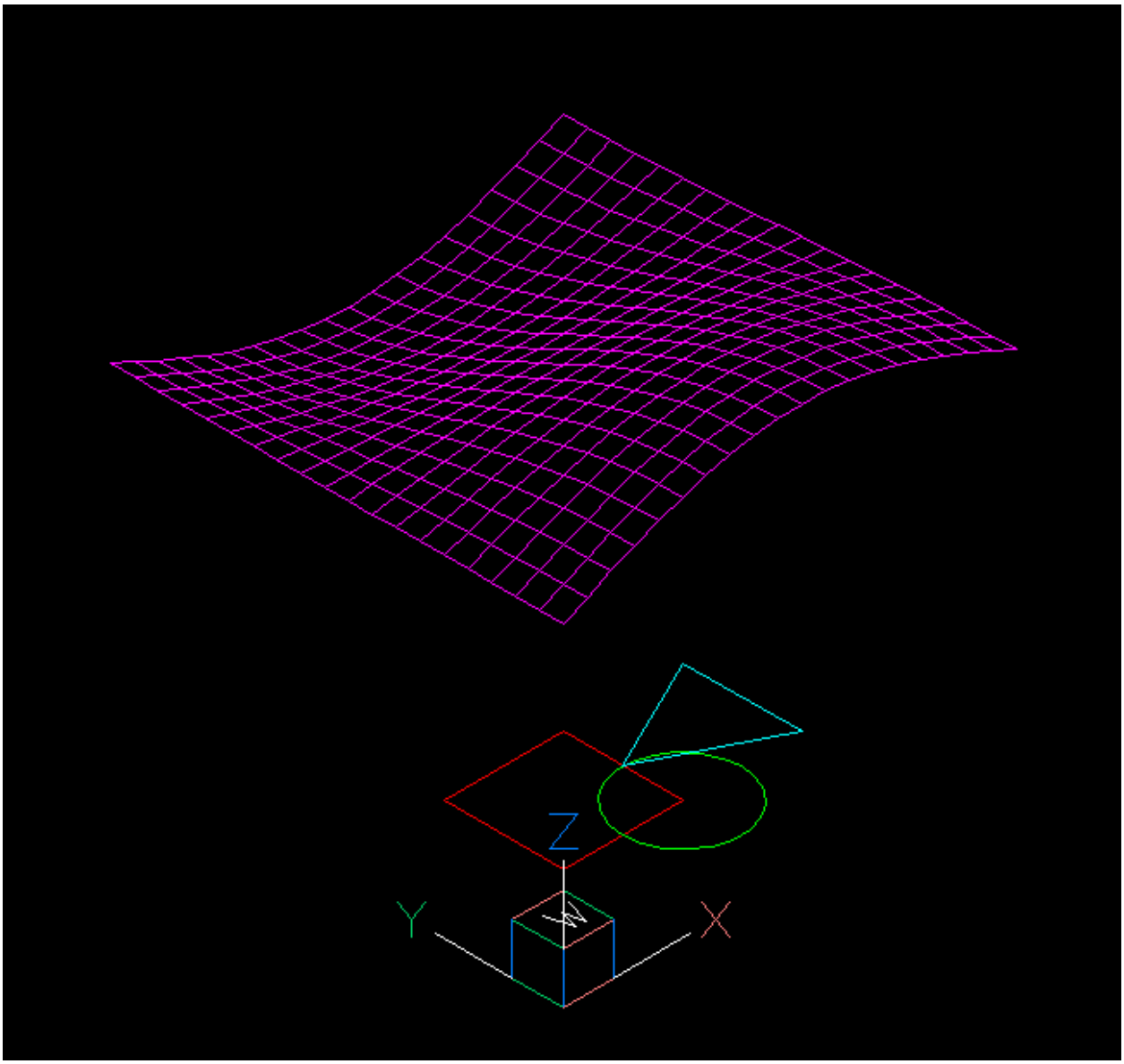
The style for a MULTILEADER is set at the *add_multileader_mtext()* and *add_multileader_block()* factory methods.

6.5.20 Tutorial for Viewports in Paperspace

This tutorial is based on the example script *viewports_in_paperspace.py*. The script creates DXF files for the version R12 and for R2000+, but the export for DXF R12 has a wrong papersize in BricsCAD and wrong margins in Autodesk DWG Trueview. I don't know why this happens and I don't waste my time to fix this.

Important: If you need paperspace layouts use DXF version R2000 or newer because the export of the page dimensions does not work for DXF R12!

The scripts creates three flat geometries in the xy-plane of the *WCS* and a 3D mesh as content of the modelspace:



Page Setup

The paperspace layout feature lacks documentation in the DXF reference, there is no information in practice on **how** it is used, so most of the information here is assumptions gathered through trail and error.

The *page_setup()* method defines the properties of the paper sheet itself. The units of the modelspace and the paperspace are not related and can even have different unit systems (imperial, meters), but to keep things simple it's recommended to use the same unit system for both spaces.

```
layout.page_setup(size=(24, 18), margins=(1, 1, 1, 1), units="inch")
```

The *size* argument defines the overall paper size in rotation mode 0, it seems to be the best practice to define the paper extents in landscape mode and rotate the paper by the *rotate* argument afterwards.

Choices for the *rotation* argument:

0	no rotation
1	90 degrees counter-clockwise
2	upside-down
3	90 degrees clockwise

The *scale* argument reflects the relationship between paper unit and drawing unit in paperspace. It's recommended to let this scale at the default value of 1:1 and draw lines and text in paperspace with the same units as you defined the paper size.

See also:

- AutoCAD: About Plotting and About Setting the Plot Scale
- BricsCAD: General Procedure for Printing

Drawing in Paperspace

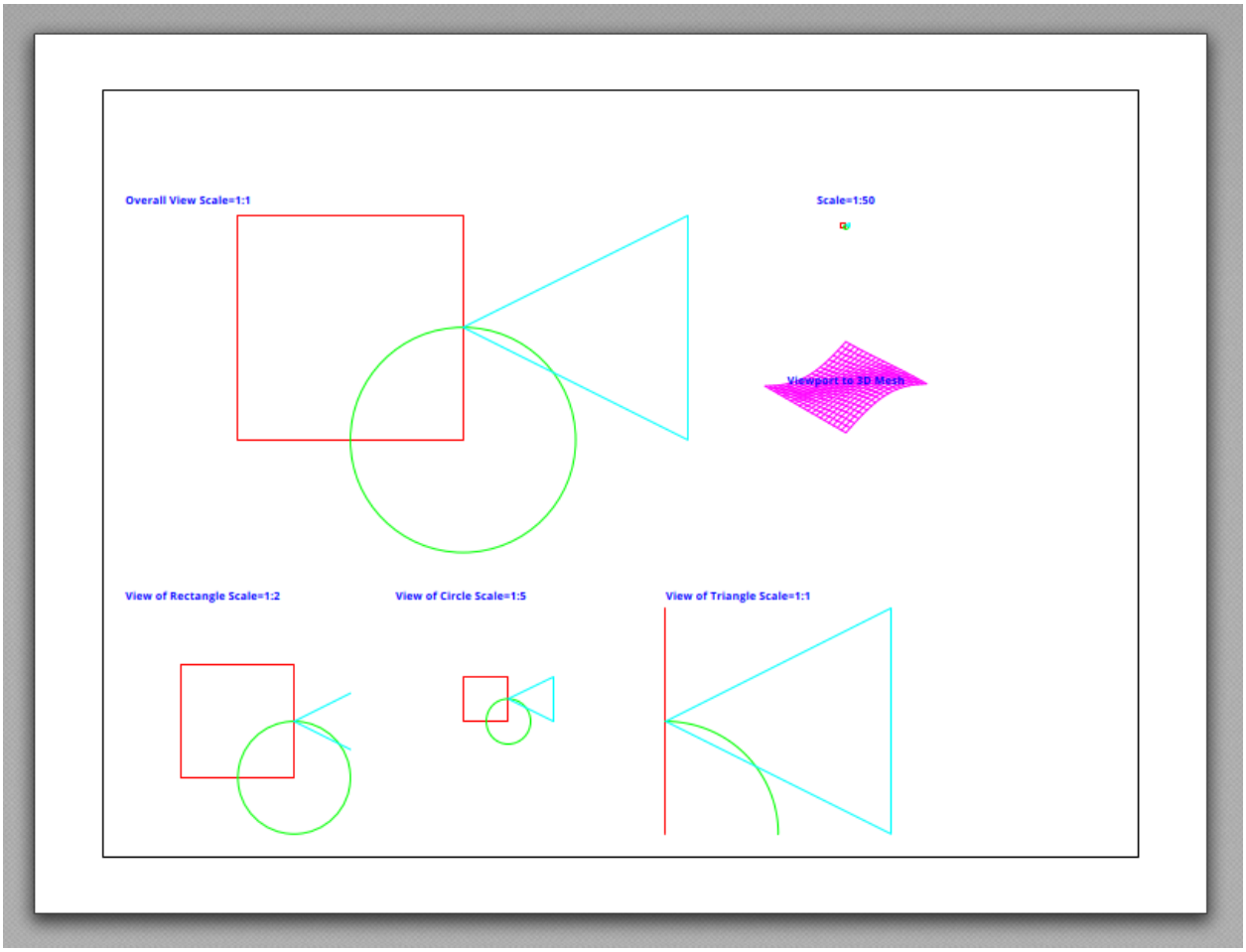
You can add DXF entities to the paperspace like to any other layout space. The coordinate origin (0, 0) is in the left bottom corner of the canvas which is the paper size minus the margins. You can draw beyond this limits but CAD applications may not print that content.

Hint: By writing this tutorial I noticed that changing the printer/plotter and the paper size does shift the layout content, because all paper sizes are defined without margins. Maybe it's preferable to set all margins to zero.

I added the helper method `page_setup()` to the `Drawing` class and an example simple_page_setup.py how to use it.

Adding Viewports

The `Viewport` entity is a window to the modelspace to display the content of the modelspace in paperspace with an arbitrary scaling and rotation. The VIEWPORT entity will be added by the factory method `add_viewport()`, the *center* argument defines the center and the *size* argument defines the width and height of the of the VIEWPORT in paperspace. The source of the modelspace to display is defined by the arguments *view_center_point* and *view_height*.



Scaling Factor

The scaling factor of the VIEWPORT is not an explicit value, the factor is defined by the relation of the VIEWPORT height of the *size* argument and the *view_height* argument.

If both values are equal the scaling is 1:1

```
paperspace.add_viewport(
    center=(14.5, 2.5),
    size=(5, 5),
    view_center_point=(12.5, 7.5),
    view_height=5,
)
```

If the *view_height* is 5x larger than the VIEWPORT height the scaling is 1:5

```
paperspace.add_viewport(
    center=(8.5, 2.5),
    size=(5, 5),
    view_center_point=(10, 5),
    view_height=25,
)
```

View Direction

The default view direction is the top down view, but can be changed to any view by the attributes *view_target_point* and *view_direction_vector* of the `dxf` namespace.

```
vp = paperspace.add_viewport(
    center=(16, 10), size=(4, 4), view_center_point=(0, 0), view_height=30
)
vp.dxf.view_target_point = (40, 40, 0)
vp.dxf.view_direction_vector = (-1, -1, 1)
```

Viewport Frame

The VIEWPORT frame (borderlines) are shown in paperspace by default. The VIEWPORT entity does not have an attribute to change this. The visibility of the VIEWPORT frame is controlled by the layer assigned to the VIEWPORT entity which is the layer "VIEWPORTS" by default in *ezdxf*. Turning off this layer hides the frames of the VIEWPORT entities on this layer, to do that the layer "VIEWPORTS" have to be created by the library user:

```
vp_layer = doc.layers.add("VIEWPORTS")
vp_layer.off()
```

Freeze Layers

Each VIEWPORT can have individual frozen layers, which means the layers are not visible in this VIEWPORT. To freeze layers in a VIEWPORT assign the names of the frozen layers as a list-like object to the `frozen_layers` attribute of the VIEWPORT entity:

```
vp.frozen_layers = ["Layer0", "Layer1"]
```

Important: AutoCAD and BricsCAD **do not crash** if the layer names do not have layer table entries and the layer names are case insensitive as all table names.

See also:

- Basic concept of *Layers*
- *`Layer`*

Override Layer Properties

Each VIEWPORT can override layer properties individually. These overrides are stored in the *`Layer`* entity and referenced by the handle of the VIEWPORT. This procedure is a bit more complex and shown in the example file viewports_override_layer_attributes.py.

1. get the *`Layer`* object
2. get the *`LayerOverrides`* object from the layer
3. override the properties of the VIEWPORT
4. commit changes

```
layer = doc.layers.get("Layer0")
override = layer.get_vp_overrides()
override.set_linetype(vp.dxf.handle, "DASHED")
override.commit()
```

Supported property overrides:

- ACI color
- true color
- transparency
- linetype
- lineweight

See also:

- Basic concept of *Layers*
- Basic concept of *AutoCAD Color Index (ACI)*
- Basic concept of *True Color*
- Basic concept of *Transparency*
- Basic concept of *Linetypes*
- Basic concept of *Lineweights*
- `Layer`
- `LayerOverrides`

6.5.21 Tutorial for OCS/UCS Usage

For OCS/UCS usage is a basic understanding of vector math required, for a brush up, watch the YouTube tutorials of 3Blue1Brown about Linear Algebra.

Second read the *Coordinate Systems* introduction please.

See also:

The free online book 3D Math Primer for Graphics and Game Development is a very good resource for learning vector math and other graphic related topics, it is easy to read for beginners and especially targeted to programmers.

For *WCS* there is not much to say as, it is what it is: the main world coordinate system, and a drawing unit can have any real world unit you want. Autodesk added some mechanism to define a scale for dimension and text entities, but because I am not an AutoCAD user, I am not familiar with it, and further more I think this is more an AutoCAD topic than a DXF topic.

Object Coordinate System (OCS)

The *OCS* is used to place planar 2D entities in 3D space. **ALL** points of a planar entity lay in the same plane, this is also true if the plane is located in 3D space by an OCS. There are three basic DXF attributes that gives a 2D entity its spatial form.

Extrusion

The extrusion vector defines the OCS, it is a normal vector to the base plane of a planar entity. This *base plane* is always located in the origin of the *WCS*. But there are some entities like `Ellipse`, which have an extrusion vector, but do not establish an OCS. For this entities the extrusion vector defines only the extrusion direction and thickness defines the extrusion distance, but all other points and directions in WCS.

Elevation

The elevation value defines the z-axis value for all points of a planar entity, this is an OCS value, and defines the distance of the entity plane from the *base plane*.

This value exists only in output from DXF versions prior to R11 as separated DXF attribute (group code 38). In DXF R12 and later, the elevation value is supplied as z-axis value of each point. But as always in DXF, this simple rule does not apply to all entities: `LWPolyline` and `Hatch` have an DXF attribute `elevation` as a 3D point, where the z-values of this point is the elevation height and the x-value and the y-value are 0.

Thickness

Defines the extrusion distance for an entity.

Note: There is a new edition of this tutorial using UCS based transformation, which are available in *ezdxf* v0.11 and later: *Tutorial for UCS Based Transformations*

This edition shows the **hard way** to accomplish the transformations by low level operations.

Placing 2D Circle in 3D Space

The colors of the system axis follow the AutoCAD standard:

- red is x-axis
- green is y-axis
- blue is z-axis

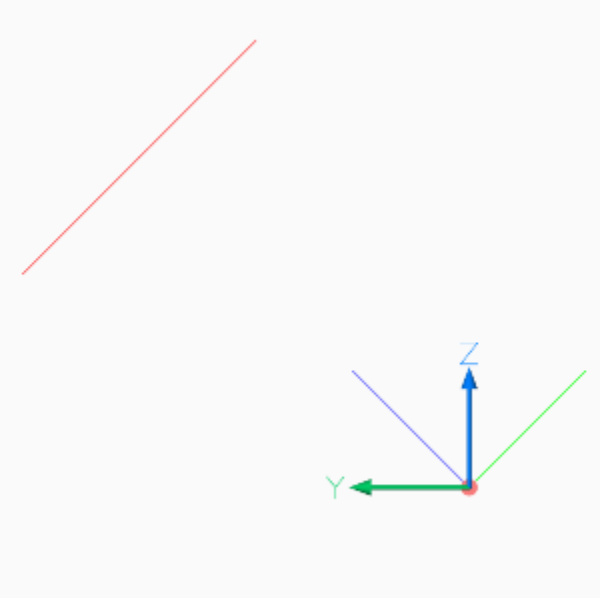
```
import ezdxf
from ezdxf.math import OCS

doc = ezdxf.new('R2010')
msp = doc.modelspace()

# For this example the OCS is rotated around x-axis about 45 degree
# OCS z-axis: x=0, y=1, z=1
# extrusion vector must not normalized here
```

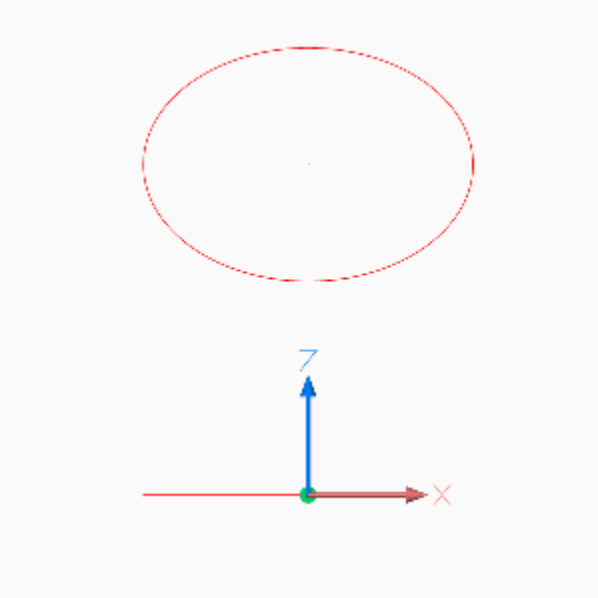
(continues on next page)

(continued from previous page)

```
ocs = OCS((0, 1, 1))
msp.add_circle(
    # You can place the 2D circle in 3D space
    # but you have to convert WCS into OCS
    center=ocs.from_wcs((0, 2, 2)),
    # center in OCS: (0.0, 0.0, 2.82842712474619)
    radius=1,
    dxfattribs={
        # here the extrusion vector should be normalized,
        # which is granted by using the ocs.uz
        'extrusion': ocs.uz,
        'color': 1,
    })
# mark center point of circle in WCS
msp.add_point((0, 2, 2), dxfattribs={'color': 1})
```

The following image shows the 2D circle in 3D space in AutoCAD *Left* and *Front* view. The blue line shows the OCS z-axis (extrusion direction), elevation is the distance from the origin to the center of the circle in this case 2.828, and you see that the x- and y-axis of the OCS and the WCS are not aligned.





Placing LWPolyline in 3D Space

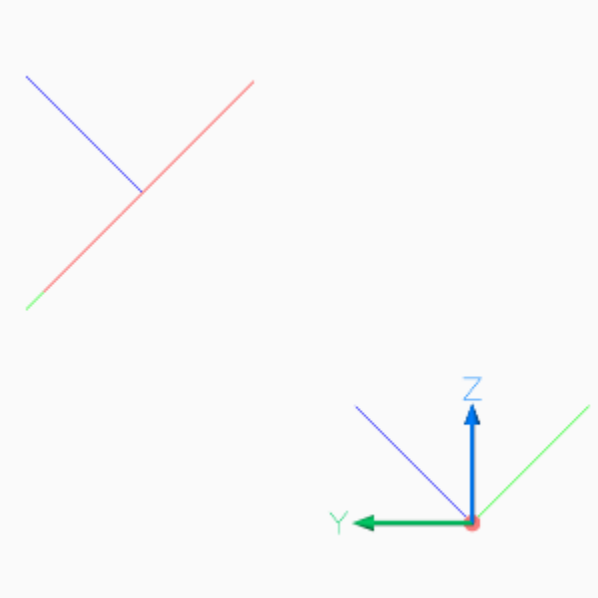
For simplicity of calculation I use the *UCS* class in this example to place a 2D pentagon in 3D space.

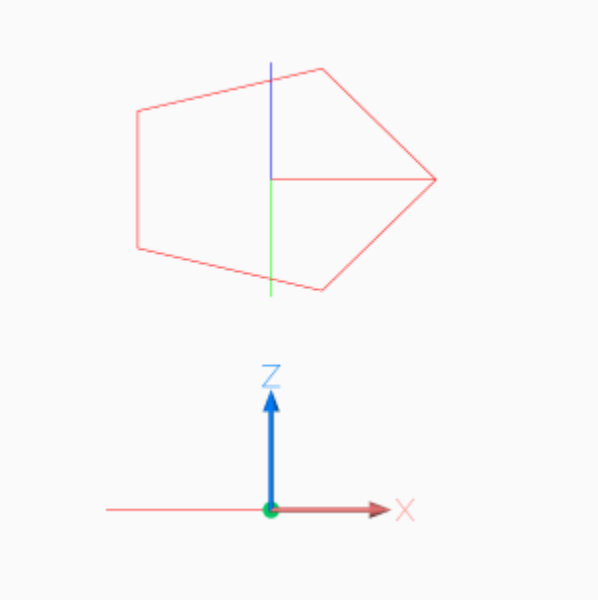
```
# The center of the pentagon should be (0, 2, 2), and the shape is
# rotated around x-axis about 45 degree, to accomplish this I use an
# UCS with z-axis (0, 1, 1) and an x-axis parallel to WCS x-axis.
ucs = UCS(
    origin=(0, 2, 2),  # center of pentagon
    ux=(1, 0, 0),  # x-axis parallel to WCS x-axis
    uz=(0, 1, 1),  # z-axis
)
# calculating corner points in local (UCS) coordinates
points = [Vec3.from_deg_angle((360 / 5) * n) for n in range(5)]
# converting UCS into OCS coordinates
ocs_points = list(ucs.points_to_ocs(points))

# LWPOLYLINE accepts only 2D points and has an separated DXF attribute elevation.
# All points have the same z-axis (elevation) in OCS!
elevation = ocs_points[0].z

msp.add_lwpolyline(
    points=ocs_points,
    format='xy',  # ignore z-axis
    close=True,
    dxfattribs={
        'elevation': elevation,
        'extrusion': ucs.uz,
        'color': 1,
    })
```

The following image shows the 2D pentagon in 3D space in AutoCAD *Left*, *Front* and *Top* view. The three lines from the center of the pentagon show the UCS, the three colored lines in the origin show the OCS, the white lines in the origin show the WCS.

The z-axis of the UCS and the OCS pointing in the same direction (extrusion direction), and the x-axis of the UCS and

the WCS pointing also in the same direction. The elevation is the distance from the origin to the center of the pentagon and all points of the pentagon have the same elevation, and you see that the y-axis of the UCS, the OCS and the WCS are not aligned.





Using UCS to Place 3D Polyline

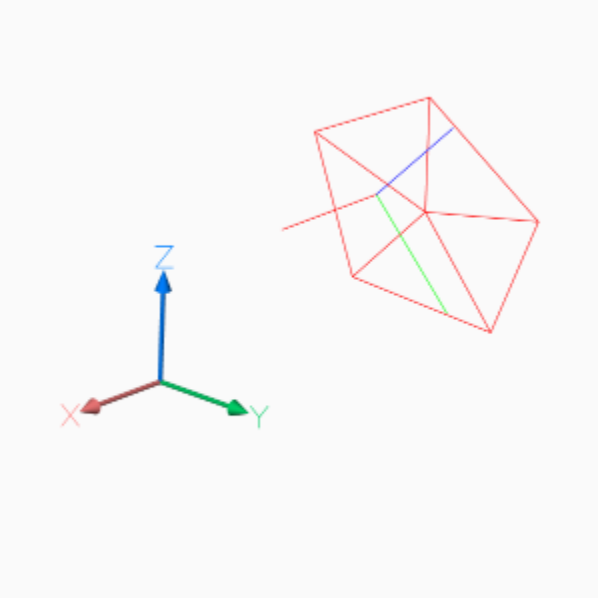
It is much simpler to use a 3D *Polyline* to create the 3D pentagon. The *UCS* class is handy for this example and all kind of 3D operations.

```
# Using an UCS simplifies 3D operations, but UCS definition can happen later
# calculating corner points in local (UCS) coordinates without Vec3 class
angle = math.radians(360 / 5)
corners_ucs = [(math.cos(angle * n), math.sin(angle * n), 0) for n in range(5)]
```

(continues on next page)

(continued from previous page)

```
# let's do some transformations
tmatrix = Matrix44.chain(  # creating a transformation matrix
    Matrix44.z_rotate(math.radians(15)),  # 1. rotation around z-axis
    Matrix44.translate(0, .333, .333),  # 2. translation
)
transformed_corners_ucs = tmatrix.transform_vertices(corners_ucs)

# transform UCS into WCS
ucs = UCS(
    origin=(0, 2, 2),  # center of pentagon
    ux=(1, 0, 0),  # x-axis parallel to WCS x-axis
    uz=(0, 1, 1),  # z-axis
)
corners_wcs = list(ucs.points_to_wcs(transformed_corners_ucs))

msp.add_polyline3d(
    points=corners_wcs,
    close=True,
)

# add lines from center to corners
center_wcs = ucs.to_wcs((0, .333, .333))
for corner in corners_wcs:
    msp.add_line(center_wcs, corner, dxfattribs={'color': 1})

ucs.render_axis(msp)
```



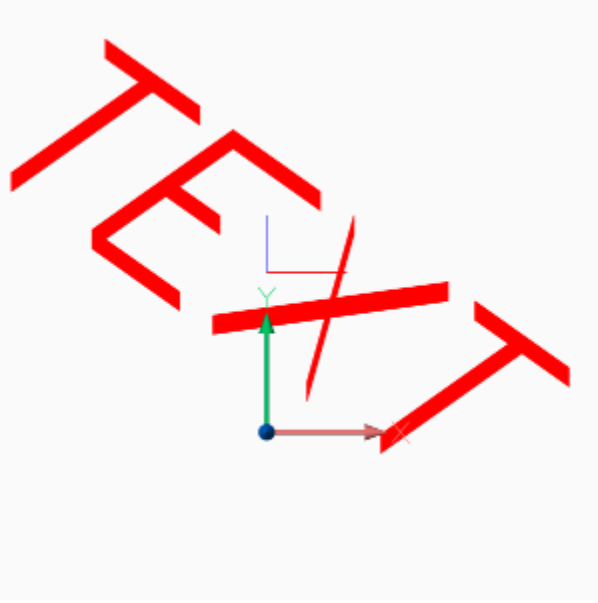
Placing 2D Text in 3D Space

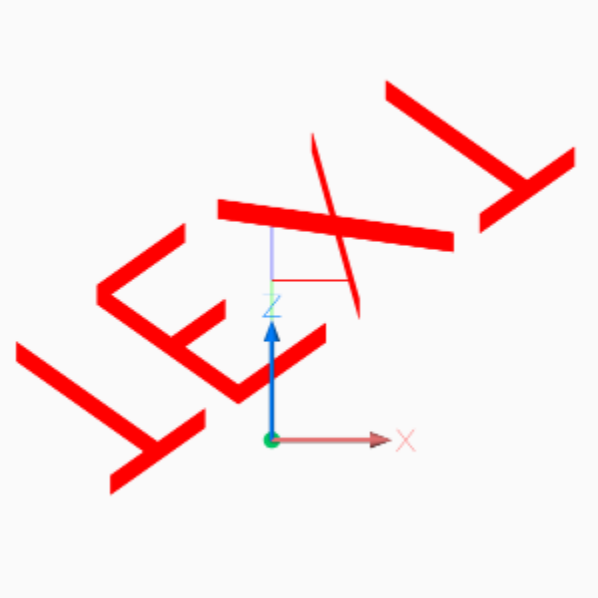
The problem of placing text in 3D space is the text rotation, which is always counter clockwise around the OCS z-axis, and `0` degree is the direction of the positive OCS x-axis, and the OCS x-axis is calculated by the *Arbitrary Axis Algorithm*.

Calculate the OCS rotation angle by converting the TEXT rotation angle (in UCS or WCS) into a vector or begin with text direction as vector, transform this direction vector into OCS and convert the OCS vector back into an angle in the OCS xy-plane (see example), this procedure is available as *`UCS.to_ocs_angle_deg()`* or `UCS.to_ocs_angle_rad()`.

AutoCAD supports thickness for the TEXT entity only for *.shx* fonts and not for true type fonts.

```
# Thickness for text works only with shx fonts not with true type fonts
doc.styles.new('TXT', dxfattribs={'font': 'romans.shx'})

ucs = UCS(origin=(0, 2, 2), ux=(1, 0, 0), uz=(0, 1, 1))
# calculation of text direction as angle in OCS:
# convert text rotation in degree into a vector in UCS
text_direction = Vec3.from_deg_angle(-45)
# transform vector into OCS and get angle of vector in xy-plane
rotation = ucs.to_ocs(text_direction).angle_deg

text = msp.add_text(
    text="TEXT",
    dxfattribs={
        # text rotation angle in degrees in OCS
        'rotation': rotation,
        'extrusion': ucs.uz,
        'thickness': .333,
        'color': 1,
        'style': 'TXT',
    })
# set text position in OCS
text.set_pos(ucs.to_ocs((0, 0, 0)), align='MIDDLE_CENTER')
```



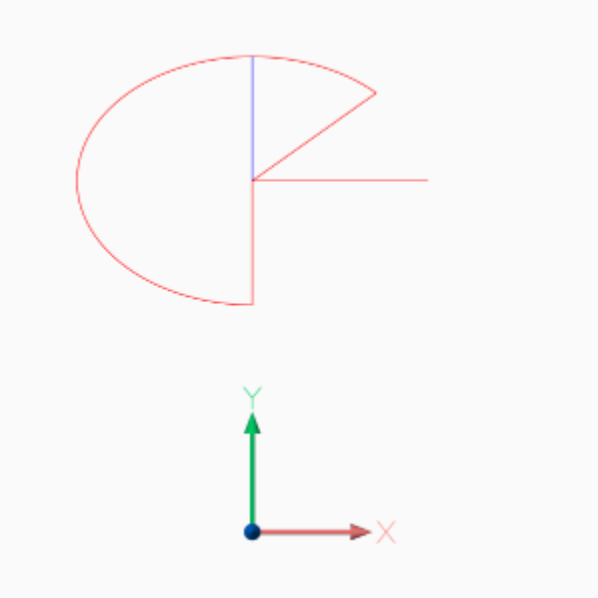


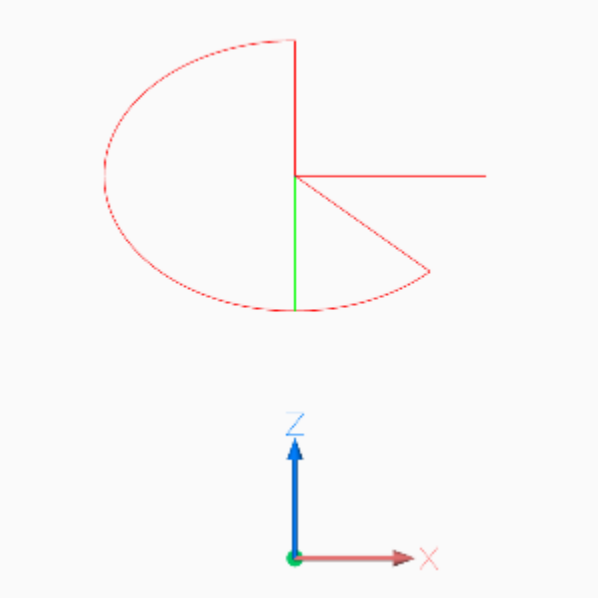
Hint: For calculating OCS angles from an UCS, be aware that 2D entities, like TEXT or ARC, are placed parallel to the xy-plane of the UCS.

Placing 2D Arc in 3D Space

Here we have the same problem as for placing text, you need the start- and end angle of the arc in degrees in the OCS, and this example also shows a shortcut for calculating the OCS angles.

```
ucs = UCS(origin=(0, 2, 2), ux=(1, 0, 0), uz=(0, 1, 1))
msp.add_arc(
    center=ucs.to_ocs((0, 0)),
    radius=1,
    start_angle=ucs.to_ocs_angle_deg(45),
    end_angle=ucs.to_ocs_angle_deg(270),
    dxfattribs={
        'extrusion': ucs.uz,
        'color': 1,
    })
center = ucs.to_wcs((0, 0))
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vec3.from_deg_angle(45)),
    dxfattribs={'color': 1},
)
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vec3.from_deg_angle(270)),
    dxfattribs={'color': 1},
)
```





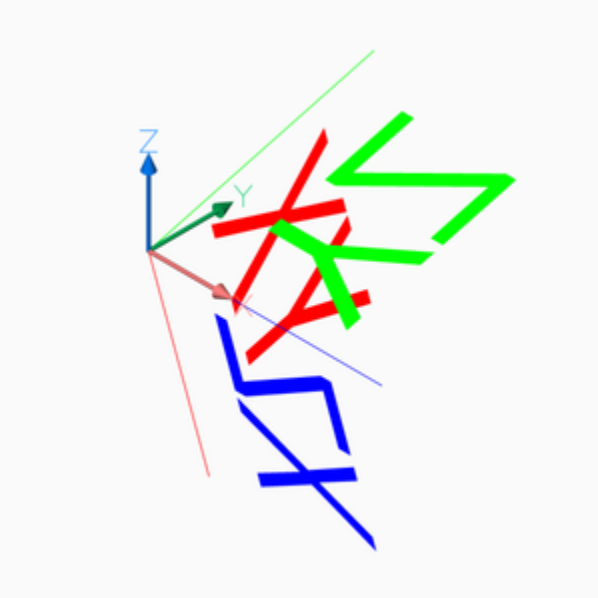
Placing Block References in 3D Space

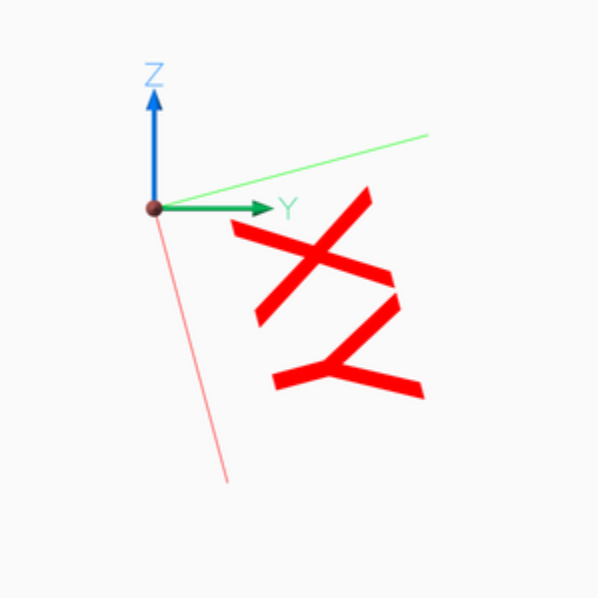
Despite the fact that block references (*Insert*) can contain true 3D entities like *Line* or *Mesh*, the *Insert* entity uses the same placing principe as *Text* or *Arc* shown in the previous chapters.

Placement by OCS coordinates and rotation about the OCS z-axis, can be achieved the same way as for generic 2D entities. The DXF attribute `Insert.dxf.rotation` rotates a block reference around the block z-axis, which is located in the `Block.dxf.base_point`. To rotate the block reference around the WCS x-axis, a transformation of the block z-axis into the WCS x-axis is required by rotating the block z-axis 90 degree counter-clockwise around y-axis by using an UCS:

This is just an excerpt of the important parts, see the whole code of insert.py at github.

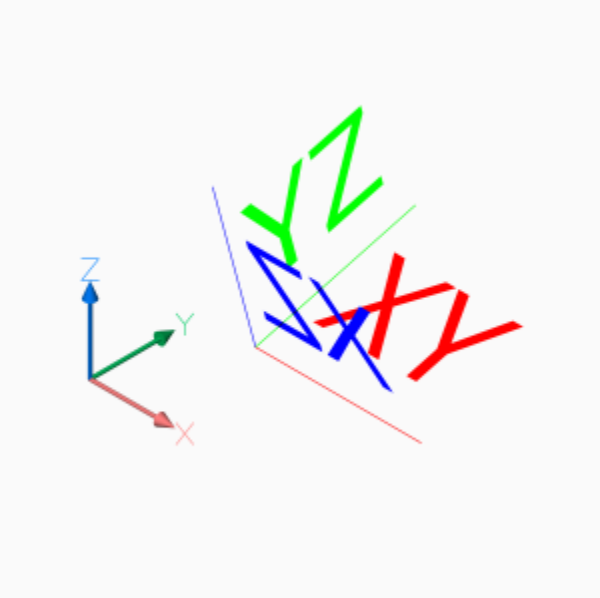
```
# rotate UCS around an arbitrary axis:
def ucs_rotation(ucs: UCS, axis: Vec3, angle: float):
    # new in ezdxf v0.11: UCS.rotate(axis, angle)
    t = Matrix44.axis_rotate(axis, math.radians(angle))
    ux, uy, uz = t.transform_vertices([ucs.ux, ucs.uy, ucs.uz])
    return UCS(origin=ucs.origin, ux=ux, uy=uy, uz=uz)



doc = ezdxf.new('R2010', setup=True)
blk = doc.blocks.new('CSYS')
setup_csys(blk)
msp = doc.modelspace()

ucs = ucs_rotation(UCS(), axis=Y_AXIS, angle=90)
# transform insert location to OCS
insert = ucs.to_ocs((0, 0, 0))
# rotation angle about the z-axis (= WCS x-axis)
rotation = ucs.to_ocs_angle_deg(15)
msp.add_blockref('CSYS', insert, dxfattribs={
   'extrusion': ucs.uz,
   'rotation': rotation,
})
```

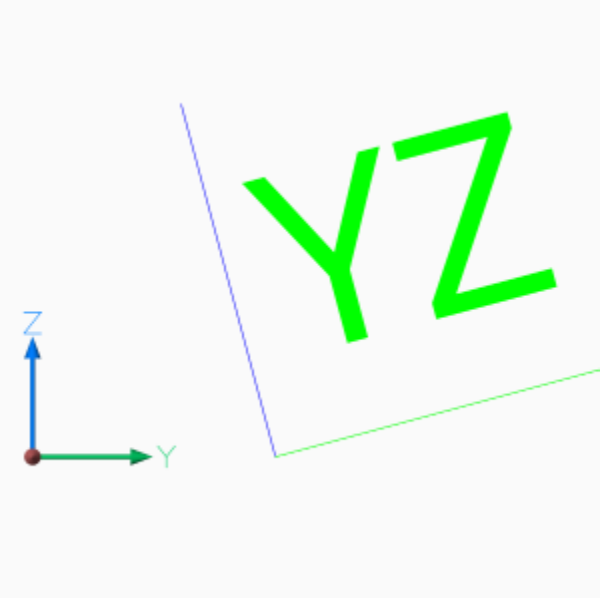




To rotate a block reference around another axis than the block z-axis, you have to find the rotated z-axis (extrusion vector) of the rotated block reference, following example rotates the block reference around the block x-axis by 15 degrees:

```
# t is a transformation matrix to rotate 15 degree around the x-axis
t = Matrix44.axis_rotate(axis=X_AXIS, angle=math.radians(15))
# transform block z-axis into new UCS z-axis (= extrusion vector)
uz = Vec3(t.transform(Z_AXIS))
# create new UCS at the insertion point, because we are rotating around the x-axis,
# ux is the same as the WCS x-axis and uz is the rotated z-axis.
ucs = UCS(origin=(1, 2, 0), ux=X_AXIS, uz=uz)
# transform insert location to OCS, block base_point=(0, 0, 0)
insert = ucs.to_ocs((0, 0, 0))
# for this case a rotation around the z-axis is not required
rotation = 0
blockref = msp.add_blockref('CSYS', insert, dxfattribs={
    'extrusion': ucs.uz,
    'rotation': rotation,
})
```

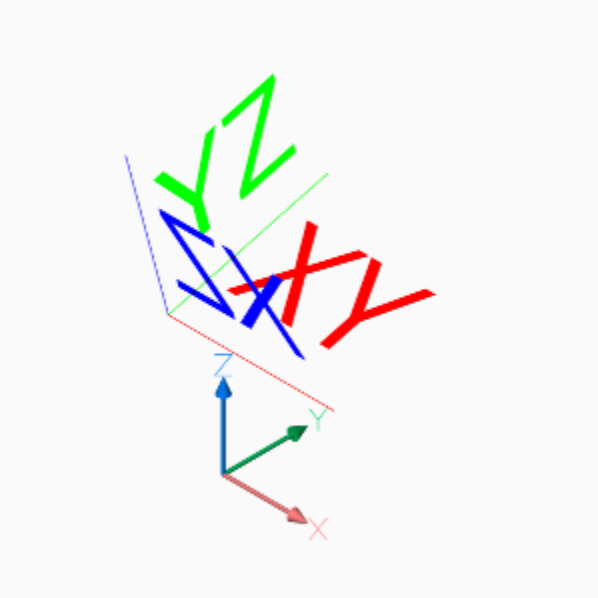


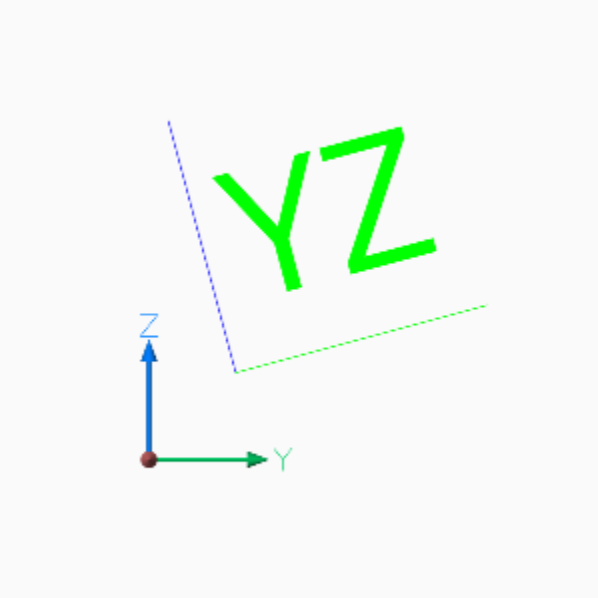


The next example shows how to translate a block references with an already established OCS:

```
# translate a block references with an established OCS
translation = Vec3(-3, -1, 1)
# get established OCS
ocs = blockref.ocs()
# get insert location in WCS
actual_wcs_location = ocs.to_wcs(blockref.dxf.insert)
# translate location
new_wcs_location = actual_wcs_location + translation
# convert WCS location to OCS location
blockref.dxf.insert = ocs.from_wcs(new_wcs_location)
```

Setting a new insert location is the same procedure without adding a translation vector, just transform the new insert location into the OCS.





The next operation is to rotate a block reference with an established OCS, rotation axis is the block y-axis, rotation angle is -90 degrees. First transform block y-axis (rotation axis) and block z-axis (extrusion vector) from OCS into WCS:

```
# rotate a block references with an established OCS around the block y-axis about 90␣
↪degree
ocs = blockref.ocs()
# convert block y-axis (= rotation axis) into WCS vector
rotation_axis = ocs.to_wcs((0, 1, 0))
# convert local z-axis (=extrusion vector) into WCS vector
local_z_axis = ocs.to_wcs((0, 0, 1))
```

Build transformation matrix and transform extrusion vector and build new UCS:

```
# build transformation matrix
t = Matrix44.axis_rotate(axis=rotation_axis, angle=math.radians(-90))
uz = t.transform(local_z_axis)
```

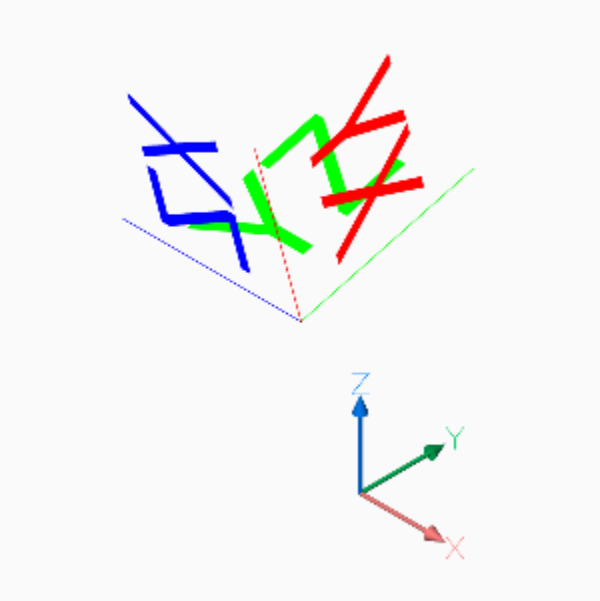
(continues on next page)

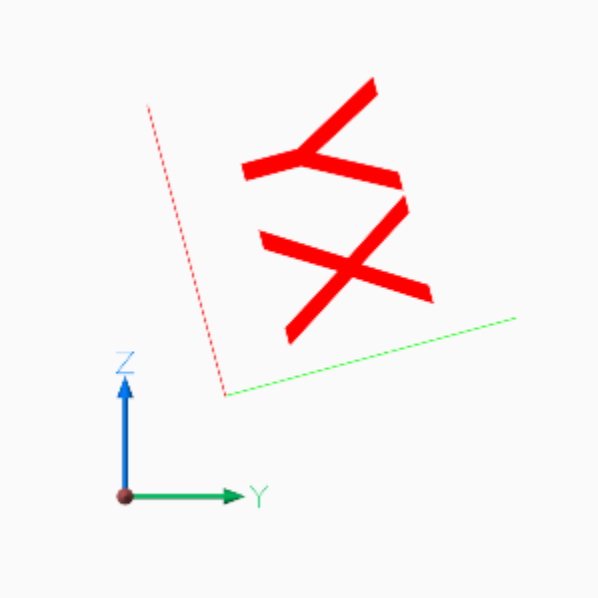
(continued from previous page)

```
uy = rotation_axis
# the block reference origin stays at the same location, no rotation needed
wcs_insert = ocs.to_wcs(blockref.dxf.insert)
# build new UCS to convert WCS locations and angles into OCS
ucs = UCS(origin=wcs_insert, uy=uy, uz=uz)
```

Set new OCS attributes, we also have to set the rotation attribute even though we do not rotate the block reference around the local z-axis, the new block x-axis (0 deg) differs from OCS x-axis and has to be adjusted:

```
# set new OCS
blockref.dxf.extrusion = ucs.uz
# set new insert
blockref.dxf.insert = ucs.to_ocs((0, 0, 0))
# set new rotation: we do not rotate the block reference around the local z-axis,
# but the new block x-axis (0 deg) differs from OCS x-axis and has to be adjusted
blockref.dxf.rotation = ucs.to_ocs_angle_deg(0)
```





And here is the point, where my math knowledge ends, for more advanced CAD operation you have to look elsewhere.

6.5.22 Tutorial for UCS Based Transformations

The *ezdxf* version v0.13 introduced a transformation interface for DXF primitives, which makes working with OCS/UCS much easier. This is a new edition of the *Tutorial for OCS/UCS Usage*. Please read the old tutorial for the basics about the OCS.

For this tutorial we don't have to worry about the OCS and the extrusion vector, this is done automatically by the `transform()` method of each DXF entity.

Placing 2D Circle in 3D Space

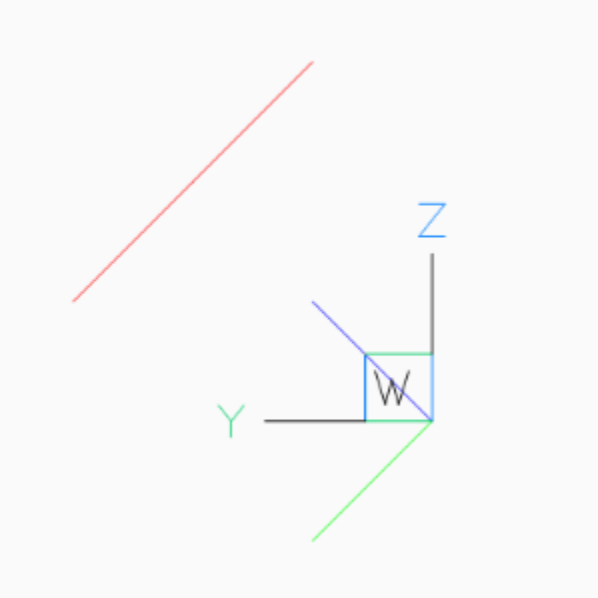
To recreate the situation of the old tutorial instantiate a new UCS and rotate it around the local x-axis. Use UCS coordinates to place the 2D CIRCLE in 3D space and transform the UCS coordinates to the WCS.

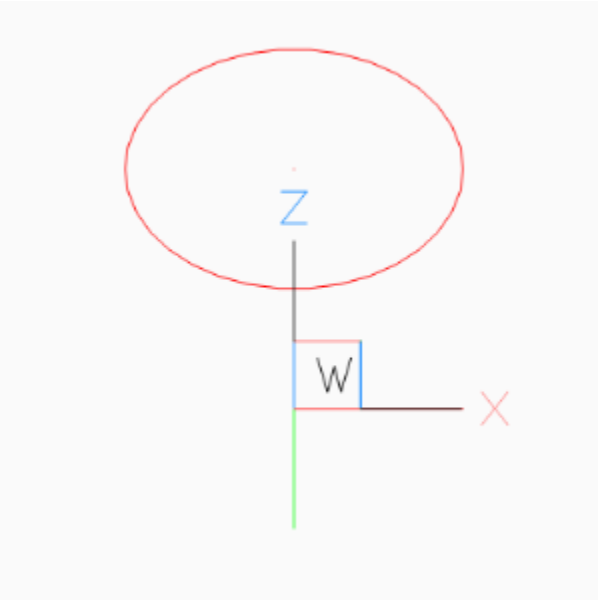
```
import math
import ezdxf
from ezdxf.math import UCS

doc = ezdxf.new('R2010')
msp = doc.modelspace()

ucs = UCS()  # New default UCS
# All rotation angles in radians, and rotation
# methods always return a new UCS.
ucs = ucs.rotate_local_x(math.radians(-45))
circle = msp.add_circle(
    # Use UCS coordinates to place the 2d circle in 3d space
    center=(0, 0, 2),
    radius=1,
    dxfattribs={'color': 1}
)
circle.transform(ucs.matrix)
```

(continues on next page)

(continued from previous page)

```
# mark center point of circle in WCS
msp.add_point((0, 0, 2), dxfattribs={'color': 1}).transform(ucs.matrix)
```



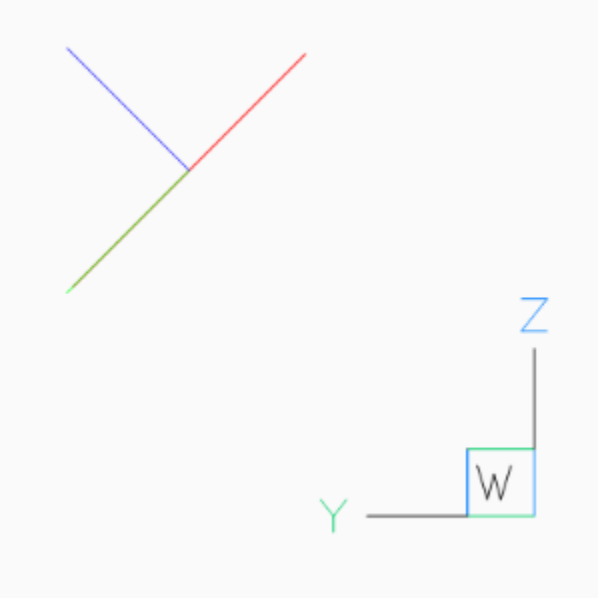


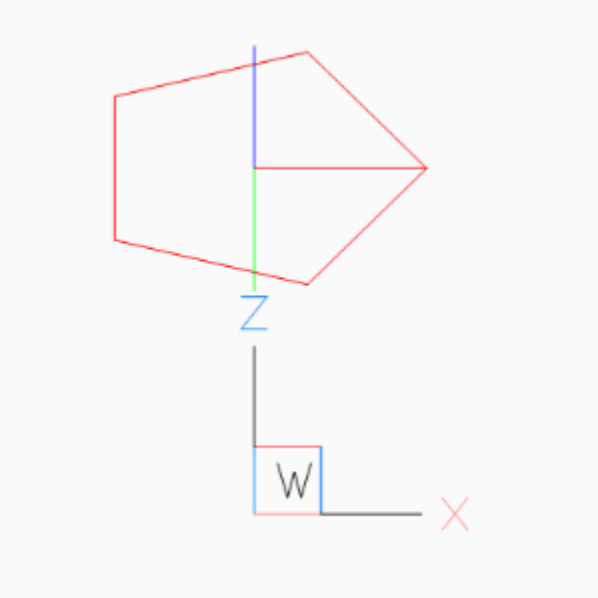
Placing LWPolyline in 3D Space

Simplified LWPOLYLINE example:

```
# The center of the pentagon should be (0, 2, 2), and the shape is
# rotated around x-axis about -45 degree
ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))

msp.add_lwpolyline(
    # calculating corner points in UCS coordinates
    points=(Vec3.from_deg_angle((360 / 5) * n) for n in range(5)),
    format='xy',  # ignore z-axis
    close=True,
    dxfattribs={
        'color': 1,
    }
```

The 2D pentagon in 3D space in BricsCAD *Left* and *Front* view.

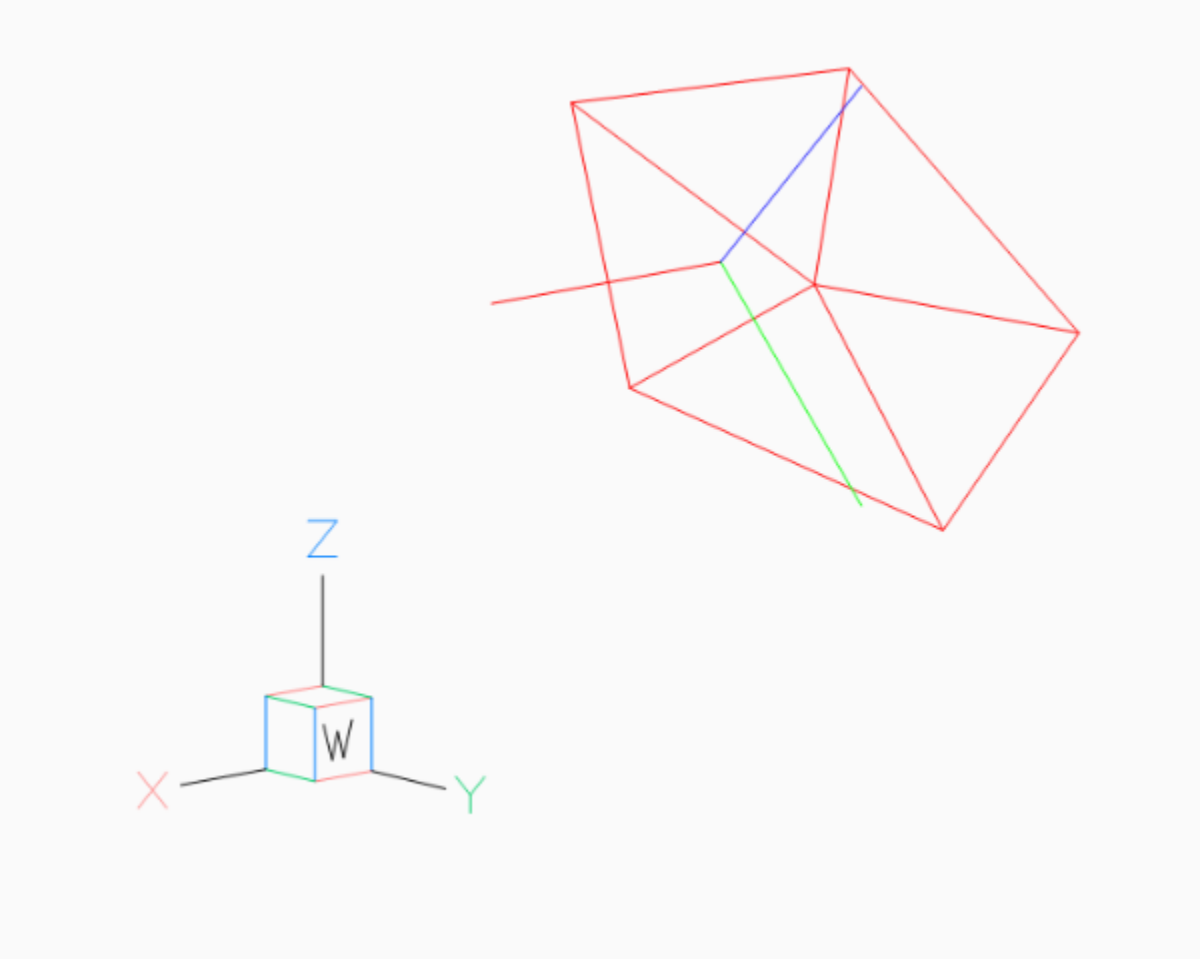




Using UCS to Place 3D Polyline

Simplified POLYLINE example: Using a first UCS to transform the POLYLINE and a second UCS to place the POLYLINE in 3D space.

```
# using an UCS simplifies 3D operations, but UCS definition can happen later
# calculating corner points in local (UCS) coordinates without Vec3 class
angle = math.radians(360 / 5)
corners_ucs = [(math.cos(angle * n), math.sin(angle * n), 0) for n in range(5)]

# let's do some transformations by UCS
transformation_ucs = UCS().rotate_local_z(math.radians(15))  # 1. rotation around z-
↪axis
transformation_ucs.shift((0, .333, .333))  # 2. translation (inplace)
corners_ucs = list(transformation_ucs.points_to_wcs(corners_ucs))

location_ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
msp.add_polyline3d(
    points=corners_ucs,
    close=True,
    dxfattribs={
        'color': 1,
    }
).transform(location_ucs.matrix)

# Add lines from the center of the POLYLINE to the corners
center_ucs = transformation_ucs.to_wcs((0, 0, 0))
for corner in corners_ucs:
    msp.add_line(
        center_ucs, corner, dxfattribs={'color': 1}
    ).transform(location_ucs.matrix)
```

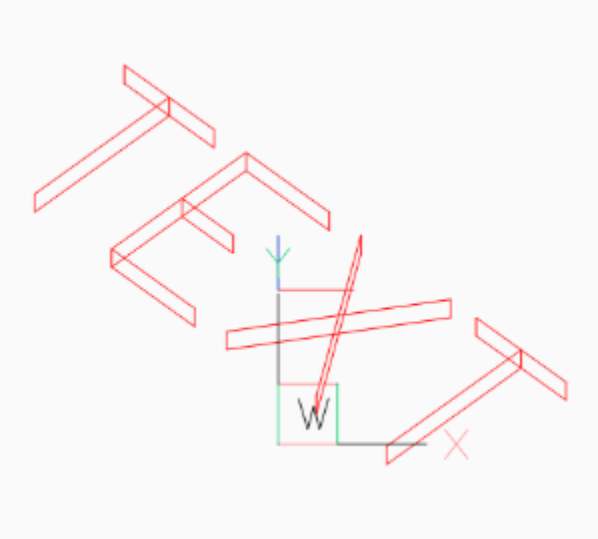


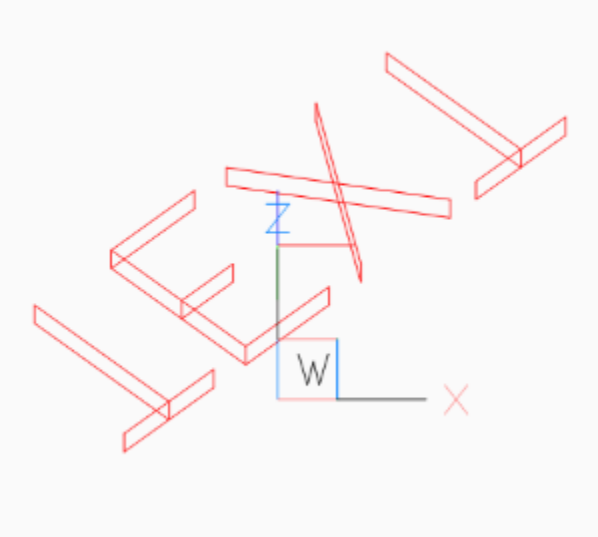
Placing 2D Text in 3D Space

The problem with the text rotation in the old tutorial disappears with the new UCS based transformation method:

AutoCAD supports thickness for the TEXT entity only for *.shx* fonts and not for true type fonts.

```
# thickness for text works only with shx fonts not with true type fonts
doc.styles.new('TXT', dxfattribs={'font': 'romans.shx'})

ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
text = msp.add_text(
    text="TEXT",
    dxfattribs={
        # text rotation angle in degrees in UCS
        'rotation': -45,
        'thickness': .333,
        'color': 1,
        'style': 'TXT',
    }
)
# set text position in UCS
text.set_pos((0, 0, 0), align='MIDDLE_CENTER')
text.transform(ucs.matrix)
```





Placing 2D Arc in 3D Space

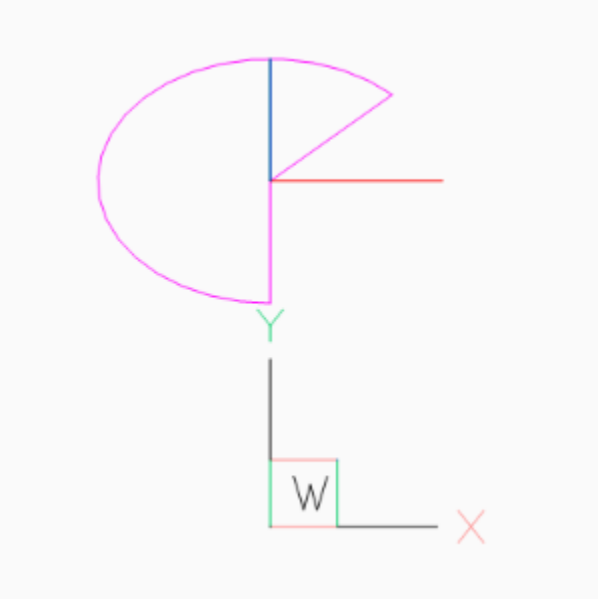
Same as for the text example, OCS angle transformation can be ignored:

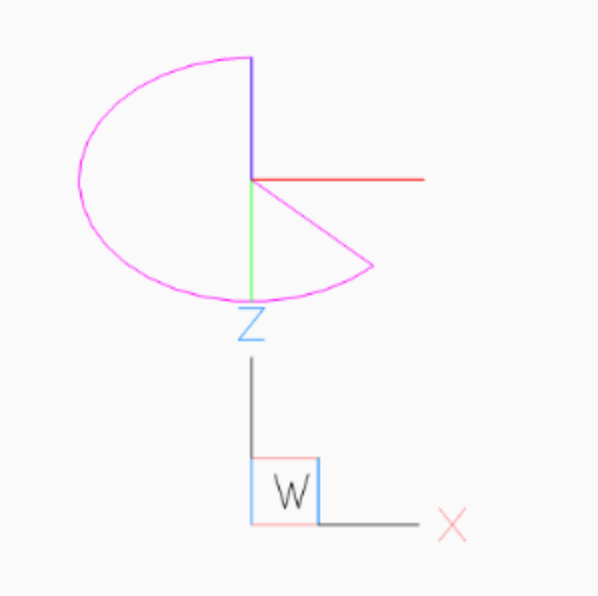
```
ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))

CENTER = (0, 0)
START_ANGLE = 45
END_ANGLE = 270

msp.add_arc(
    center=CENTER,
    radius=1,
    start_angle=START_ANGLE,
    end_angle=END_ANGLE,
    dxfattribs={'color': 6},
).transform(ucs.matrix)
```

(continues on next page)

(continued from previous page)

```
msp.add_line(
    start=CENTER,
    end=Vec3.from_deg_angle(START_ANGLE),
    dxfattribs={'color': 6},
).transform(ucs.matrix)

msp.add_line(
    start=CENTER,
    end=Vec3.from_deg_angle(END_ANGLE),
    dxfattribs={'color': 6},
).transform(ucs.matrix)
```





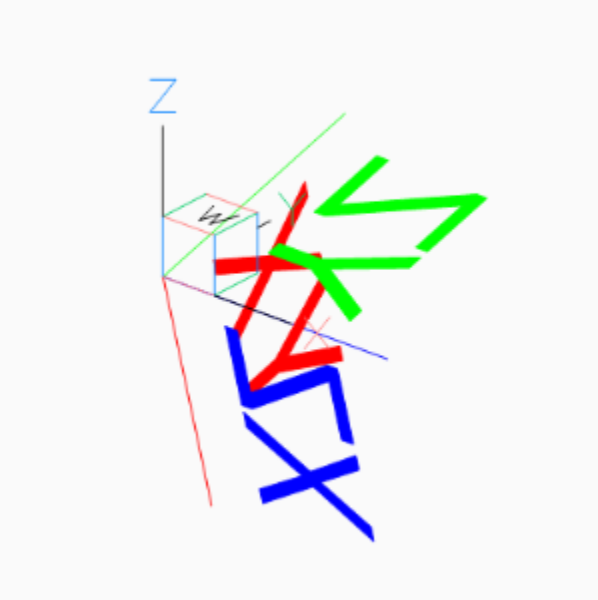
Placing Block References in 3D Space

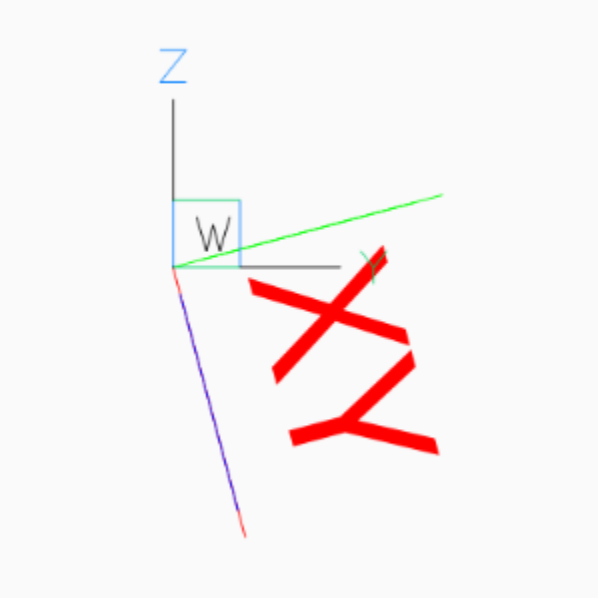
Despite the fact that block references (INSERT) can contain true 3D entities like LINE or MESH, the INSERT entity uses the same placing principe as TEXT or ARC shown in the previous sections.

To rotate the block reference 15 degrees around the WCS x-axis, we place the block reference in the origin of the UCS, and rotate the UCS 90 degrees around its local y-axis, to align the UCS z-axis with the WCS x-axis:

This is just an excerpt of the important parts, see the whole code of insert.py at github.

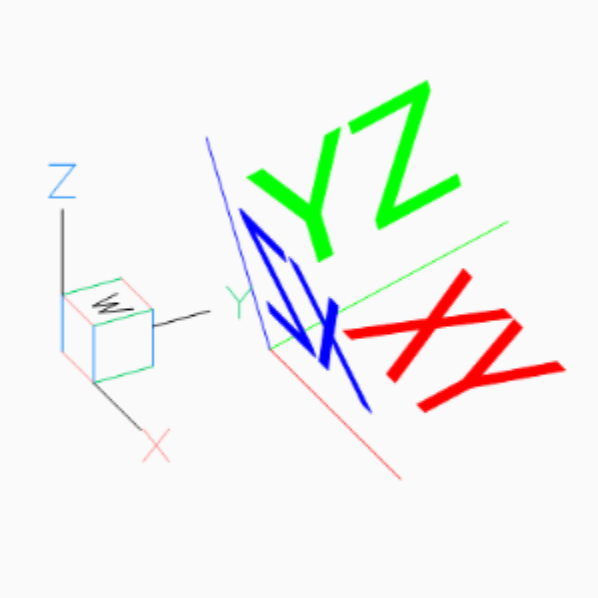
```
doc = ezdxf.new('R2010', setup=True)
blk = doc.blocks.new('CSYS')
setup_csys(blk)
msp = doc.modelspace()

ucs = UCS().rotate_local_y(angle=math.radians(90))
msp.add_blockref(
    'CSYS',
    insert=(0, 0),
    # rotation around the block z-axis (= WCS x-axis)
    dxfattribs={'rotation': 15},
).transform(ucs.matrix)
```

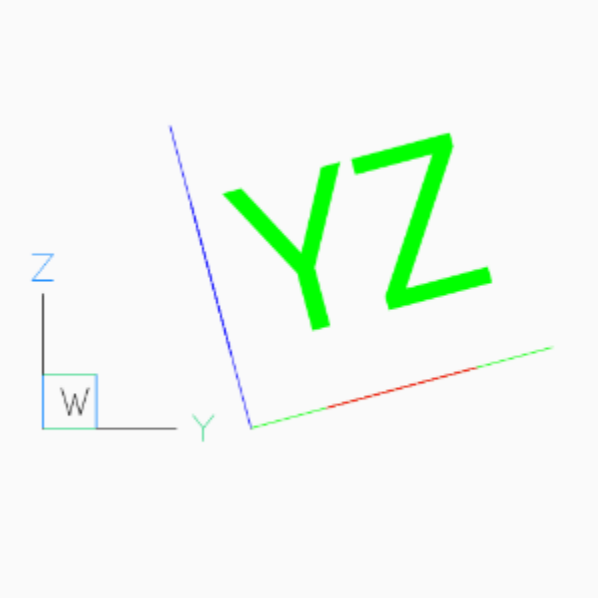




A more simple approach is to ignore the `rotate` attribute at all and just rotate the UCS. To rotate a block reference around any axis rather than the block z-axis, rotate the UCS into the desired position. The following example rotates the block reference around the block x-axis by 15 degrees:

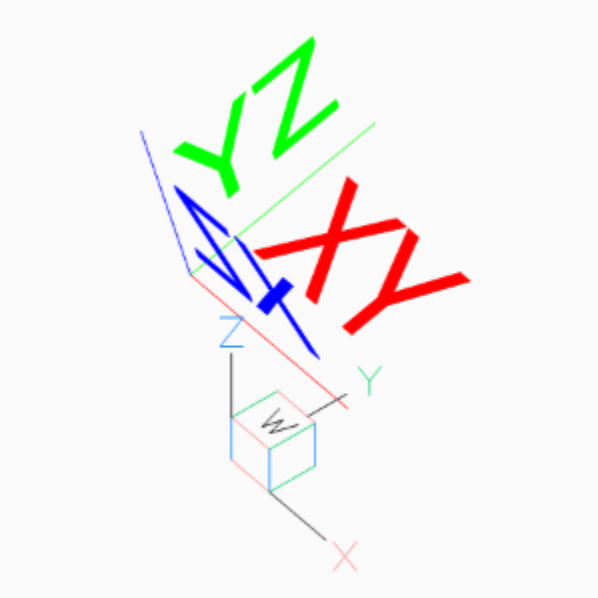
```
ucs = UCS(origin=(1, 2, 0)).rotate_local_x(math.radians(15))
blockref = msp.add_blockref('CSYS', insert=(0, 0, 0))
blockref.transform(ucs.matrix)
```





The next example shows how to translate a block references with an already established OCS:

```
# New UCS at the translated location, axis aligned to the WCS
ucs = UCS((-3, -1, 1))
# Transform an already placed block reference, including
# the transformation of the established OCS.
blockref.transform(ucs.matrix)
```

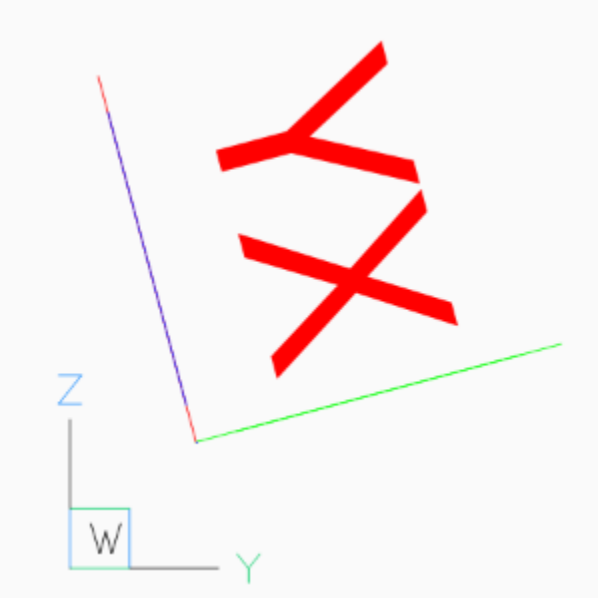


The next operation is to rotate a block reference with an established OCS, rotation axis is the block y-axis, rotation angle is -90 degrees. The idea is to create an UCS in the origin of the already placed block reference, UCS axis aligned to the block axis and resetting the block reference parameters for a new WCS transformation.

```
# Get UCS at the block reference insert location, UCS axis aligned
# to the block axis.
ucs = blockref.ucs()
# Rotate UCS around the local y-axis.
ucs = ucs.rotate_local_y(math.radians(-90))
```

Reset block reference parameters, this places the block reference in the UCS origin and aligns the block axis to the UCS axis, now we do a new transformation from UCS to WCS:

```
# Reset block reference parameters to place block reference in
# UCS origin, without any rotation and OCS.
blockref.reset_transformation()

# Transform block reference from UCS to WCS
blockref.transform(ucs.matrix)
```



6.5.23 Tutorial for Linear Dimensions

The `Dimension` entity is the generic entity for all dimension types, but unfortunately AutoCAD is **not willing** to show a dimension line defined only by this dimension entity, it also needs an anonymous block which contains the dimension line shape constructed by DXF primitives like LINE and TEXT entities, this representation is called the dimension line *rendering* in this documentation, beside the fact that this is not a real graphical rendering. BricsCAD is a much more friendly CAD application, which do show the dimension entity without the graphical rendering as block, which was very useful for testing, because there is no documentation how to apply all the dimension style variables (more than 80). This seems to be the reason why dimension lines are rendered so differently by many CAD application.

Don't expect to get the same rendering results by *ezdxf* as you get from AutoCAD. *Ezdxf* tries to be as close to the results rendered by BricsCAD, but it is not possible to implement all the various combinations of dimension style parameters, which often affect one another.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Text rendering is another problem, because *ezdxf* has no real rendering engine. Some font properties, like the real text width, which is only available to *ezdxf* if the *Matplotlib* package is installed and this value may also vary slightly for different CAD applications. Without access to the *Matplotlib* package the text properties in *ezdxf* are based on an abstract monospaced font and are bigger than required by true type fonts.

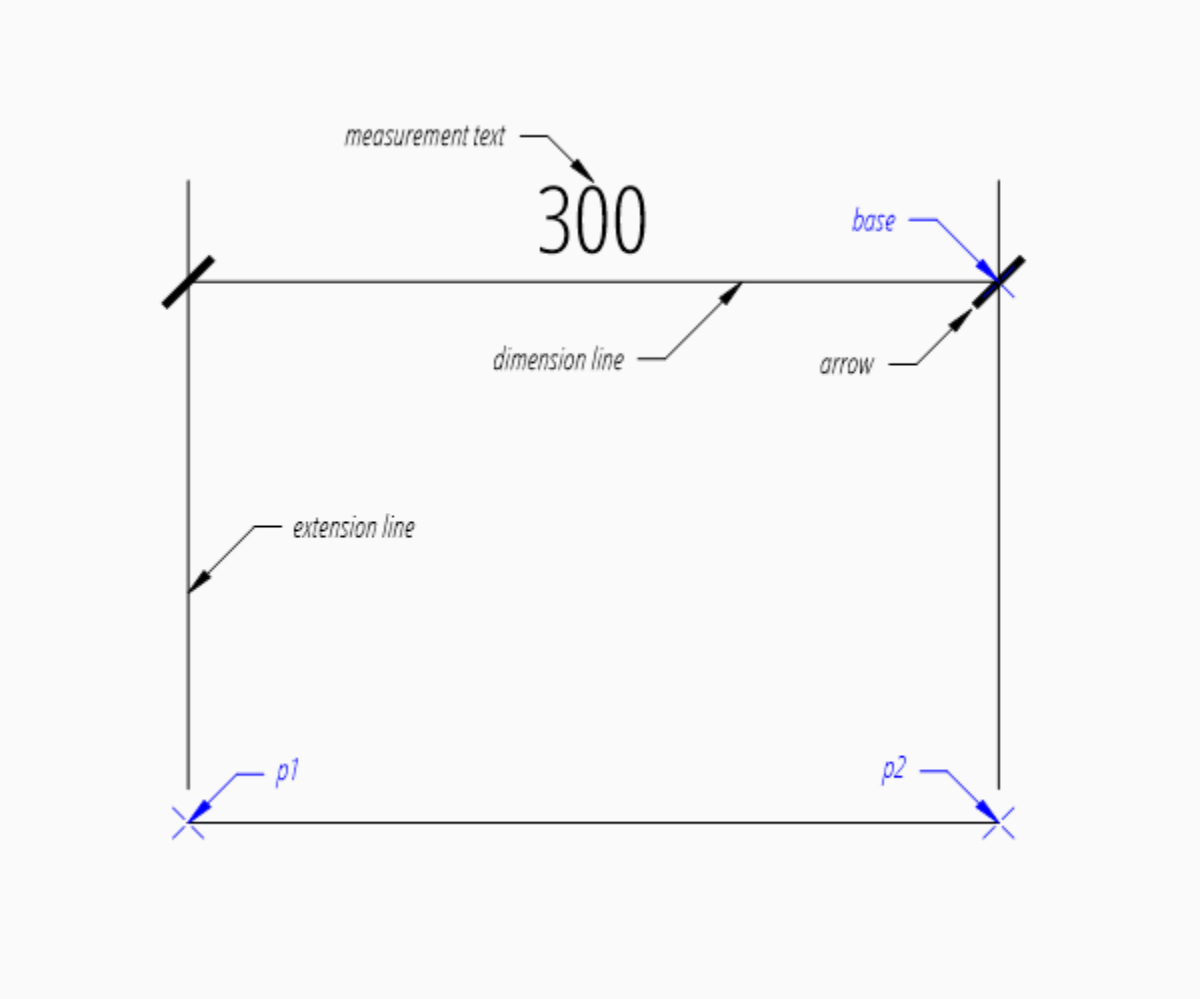
Not all DIMENSION and DIMSTYLE features are supported by all DXF versions, especially DXF R12 does not support many features, but in this case the required rendering of dimension lines is an advantage, because if the application just shows the rendered block, all features which can be used in DXF R12 will be displayed, but these features will disappear if the dimension line will be edited in the CAD application. *Ezdxf* writes only the supported DIMVARS of the used DXF version to avoid invalid DXF files. So it is not that critical to know all the supported features of a DXF version, except for limits and tolerances, *ezdxf* uses the advanced features of the MTEXT entity to create limits and tolerances and therefore they are not supported (displayed) in DXF R12 files.

See also:

- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file standards.py shows how to create your own DIMSTYLES.
- The Script dimension_linear.py shows examples for linear dimensions.

Horizontal Dimension

```
import ezdxf

# Create a DXF R2010 document:
# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new("R2010", setup=True)

# Add new dimension entities to the modelspace:
msp = doc.modelspace()

# Add a LINE entity for visualization, not required to create the DIMENSION
# entity:
msp.add_line((0, 0), (3, 0))

# Add a horizontal linear DIMENSION entity:
dim = msp.add_linear_dim(
    base=(3, 2),  # location of the dimension line
    p1=(0, 0),  # 1st measurement point
    p2=(3, 0),  # 2nd measurement point
    dimstyle="EZDXF",  # default dimension style
)

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
dim.render()
doc.saveas("dim_linear_horiz.dxf")
```



The example above creates a horizontal `Dimension` entity. The default dimension style "EZDXF" is defined as:

- 1 drawing unit = 1m
- measurement text height = 0.25 (drawing scale = 1:100)
- the length factor `dimlfac` = 100, which creates a measurement text in cm.
- arrow is "ARCHTICK", arrow size `dimasz` = 0.175

Every dimension style which does not exist will be replaced by the dimension style "Standard" at DXF export by `save()` or `saveas()` (e.g. dimension style setup was not initiated).

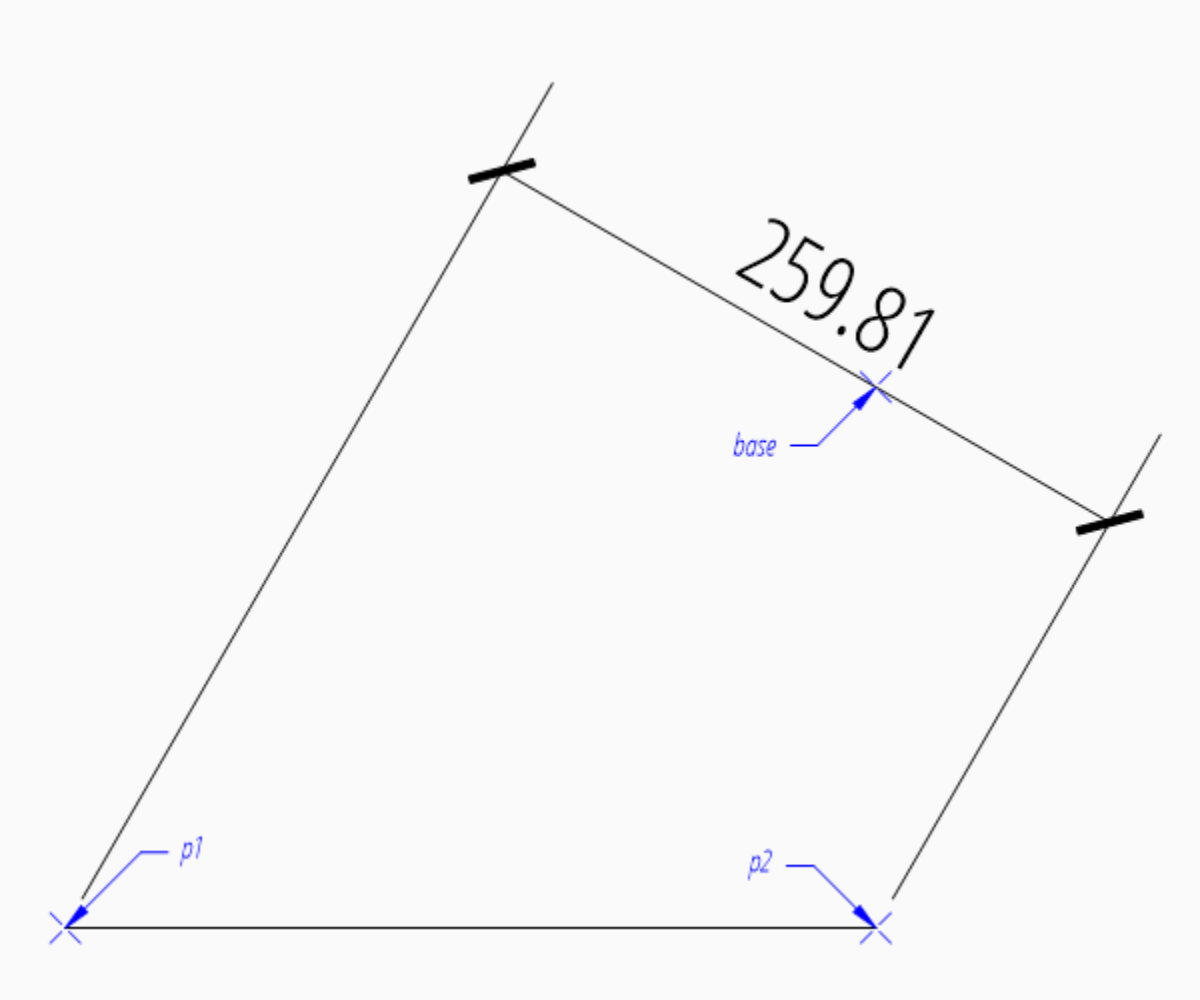
The *base* point defines the location of the dimension line, *ezdxf* accepts any point on the dimension line, the point *p1* defines the start point of the first extension line, which also defines the first measurement point and the point *p2* defines the start point of the second extension line, which also defines the second measurement point.

The return value *dim* is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as attribute `dim.dimension`.

Vertical and Rotated Dimension

Argument *angle* defines the angle of the dimension line in relation to the x-axis of the WCS or UCS, measurement is the distance between first and second measurement point in direction of *angle*.

```
# assignment to dim is not necessary, if no additional processing happens
msp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0), angle=-30).render()
doc.saveas("dim_linear_rotated.dxf")
```

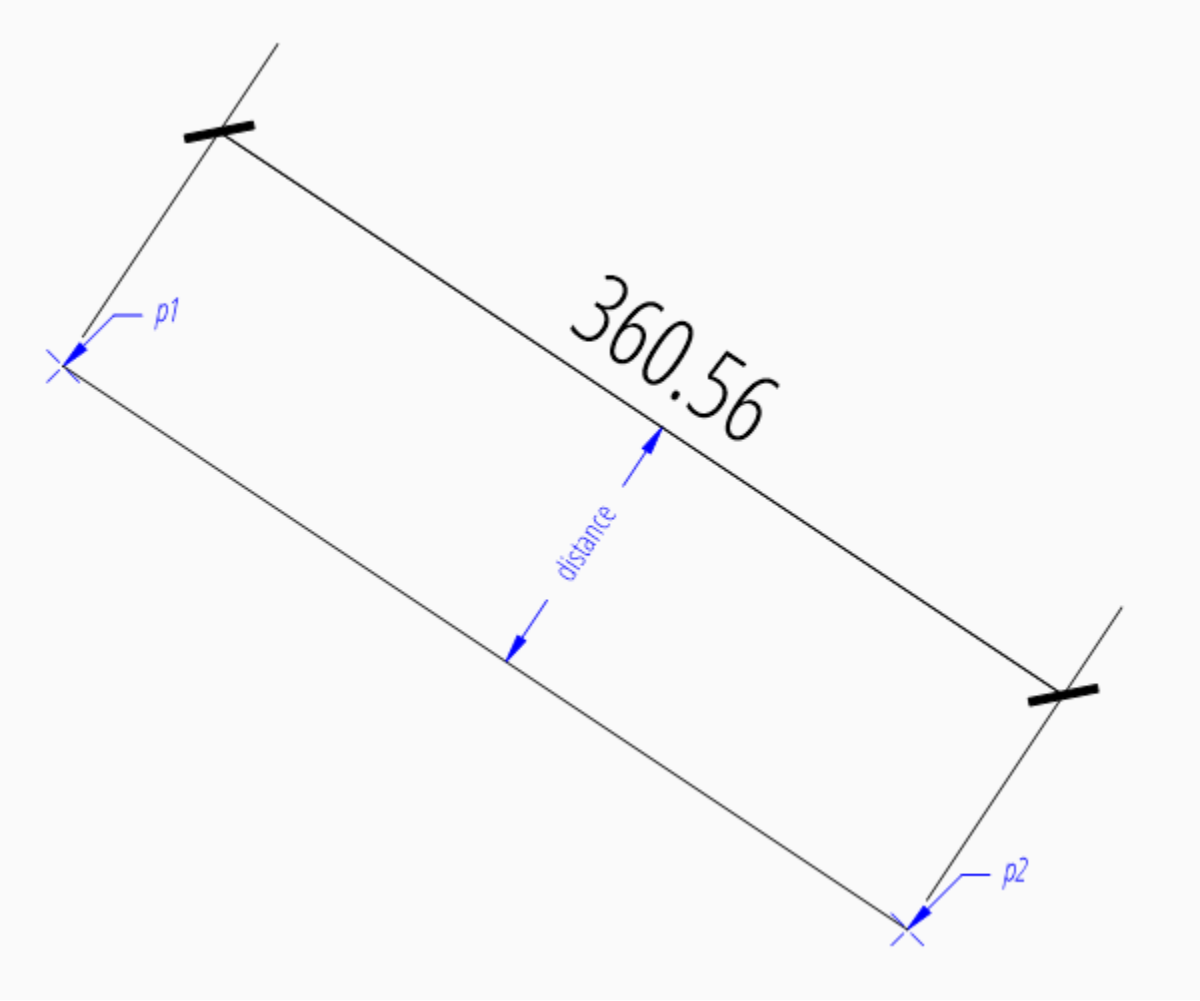


For a vertical dimension set argument *angle* to 90 degree, but in this example the vertical distance would be 0.

Aligned Dimension

An aligned dimension line is parallel to the line defined by the definition points *p1* and *p2*. The placement of the dimension line is defined by the argument *distance*, which is the distance between the definition line and the dimension line. The *distance* of the dimension line is orthogonal to the base line in counter clockwise orientation.

```
msp.add_line((0, 2), (3, 0))
dim = msp.add_aligned_dim(p1=(0, 2), p2=(3, 0), distance=1)
doc.saveas("dim_linear_aligned.dxf")
```



Dimension Style Override

Many dimension styling options are defined by the associated `DimStyle` entity. But often you wanna change just a few settings without creating a new dimension style, therefore the DXF format has a protocol to store this changed settings in the dimension entity itself. This protocol is supported by *ezdxf* and every factory function which creates dimension entities supports the *override* argument. This *override* argument is a simple Python dictionary (e.g. `override = {"dimtad": 4}`, place measurement text below dimension line).

The overriding protocol is managed by the `DimStyleOverride` object, which is returned by the most dimension factory functions.

Placing Measurement Text

The default location of the measurement text depends on various *DimStyle* parameters and is applied if no user defined text location is defined.

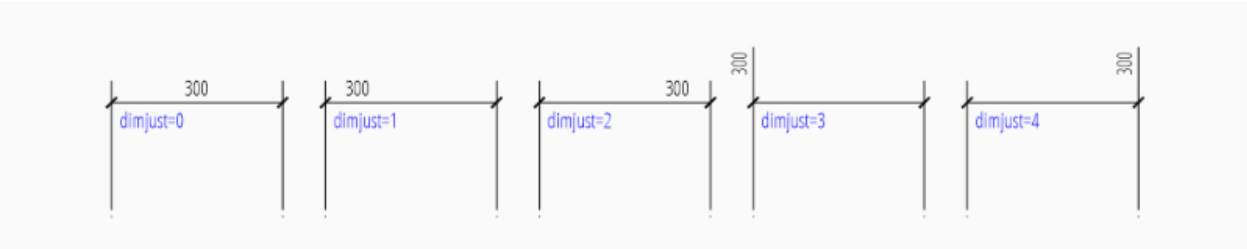
Default Text Locations

"Horizontal direction" means in direction of the dimension line and "vertical direction" means perpendicular to the dimension line direction.

The **"horizontal"** location of the measurement text is defined by *dimjust*:

0	Center of dimension line
1	Left side of the dimension line, near first extension line
2	Right side of the dimension line, near second extension line
3	Over first extension line
4	Over second extension line

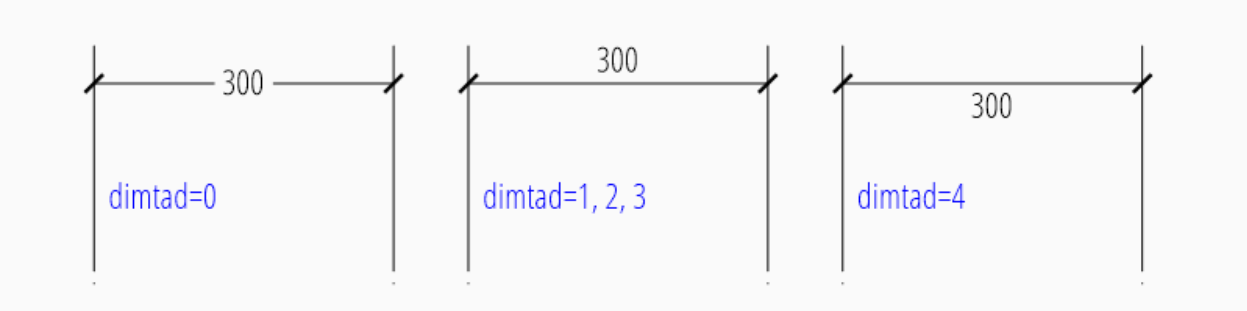
```
msp.add_linear_dim(
    base=(3, 2), p1=(0, 0), p2=(3, 0), override={"dimjust": 1}
).render()
```



The **"vertical"** location of the measurement text relative to the dimension line is defined by *dimtad*:

0	Center, it is possible to adjust the vertical location by *dimtvp*
1	Above
2	Outside, handled like *Above* by *ezdxf*
3	JIS, handled like *Above* by *ezdxf*
4	Below

```
msp.add_linear_dim(
    base=(3, 2), p1=(0, 0), p2=(3, 0), override={"dimtad": 4}
).render()
```



The distance between text and dimension line is defined by *dimgap*.

The *DimStyleOverride* object has a method *set_text_align()* to set the default text location in an easy way, this is also the reason for the 2 step creation process of dimension entities:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0))
dim.set_text_align(halign="left", valign="center")
dim.render()
```

halign	“left”, “right”, “center”, “above1”, “above2”
valign	“above”, “center”, “below”

Run function `example_for_all_text_placings_R2007()` in the example script dimension_linear.py to create a DXF file with all text placings supported by *ezdxf*.

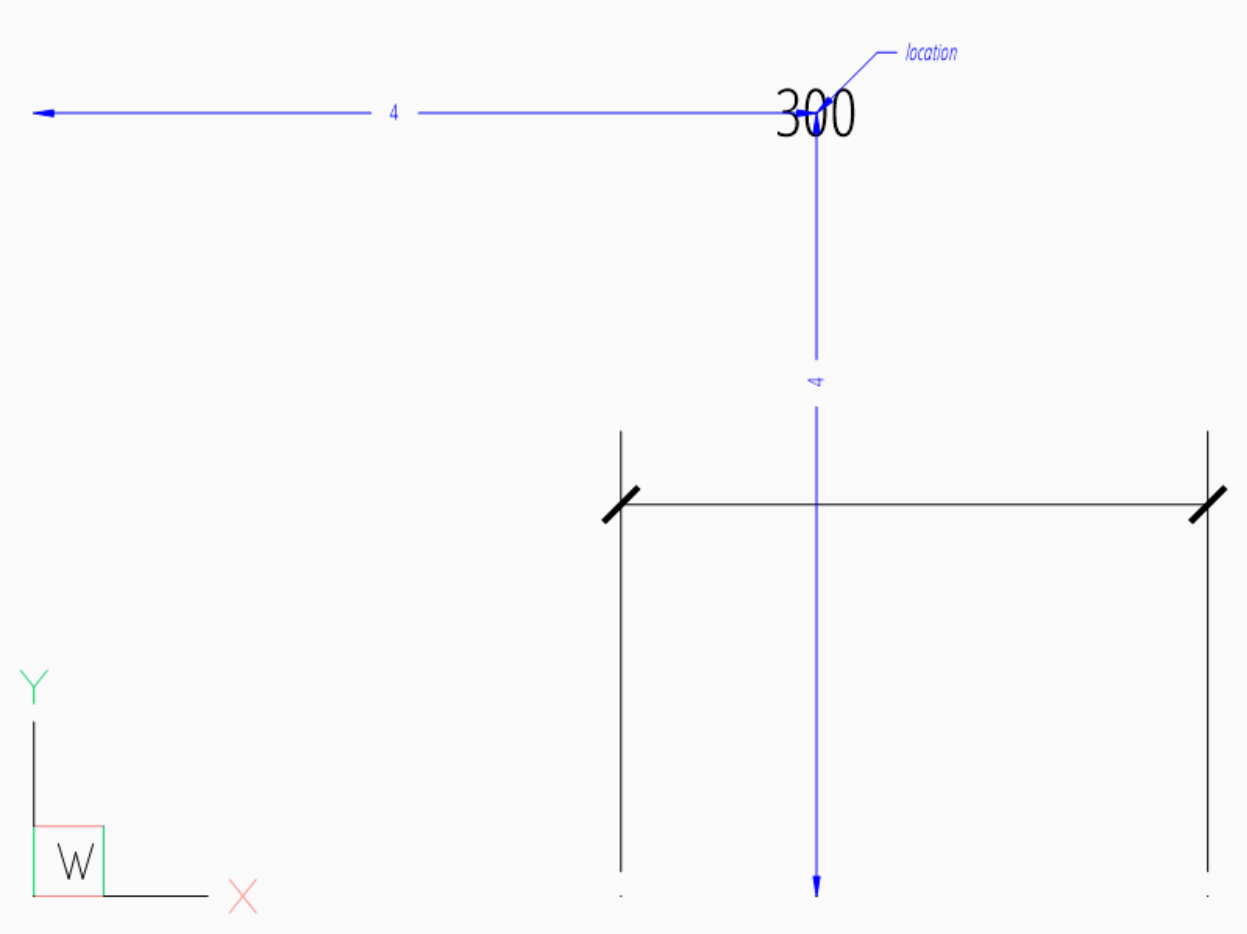
User Defined Text Locations

Beside the default location, it is possible to locate the measurement text freely.

Location Relative to Origin

The user defined text location can be set by the argument *location* in most dimension factory functions and always references the midpoint of the measurement text:

```
msp.add_linear_dim(
    base=(3, 2), p1=(3, 0), p2=(6, 0), location=(4, 4)
).render()
```

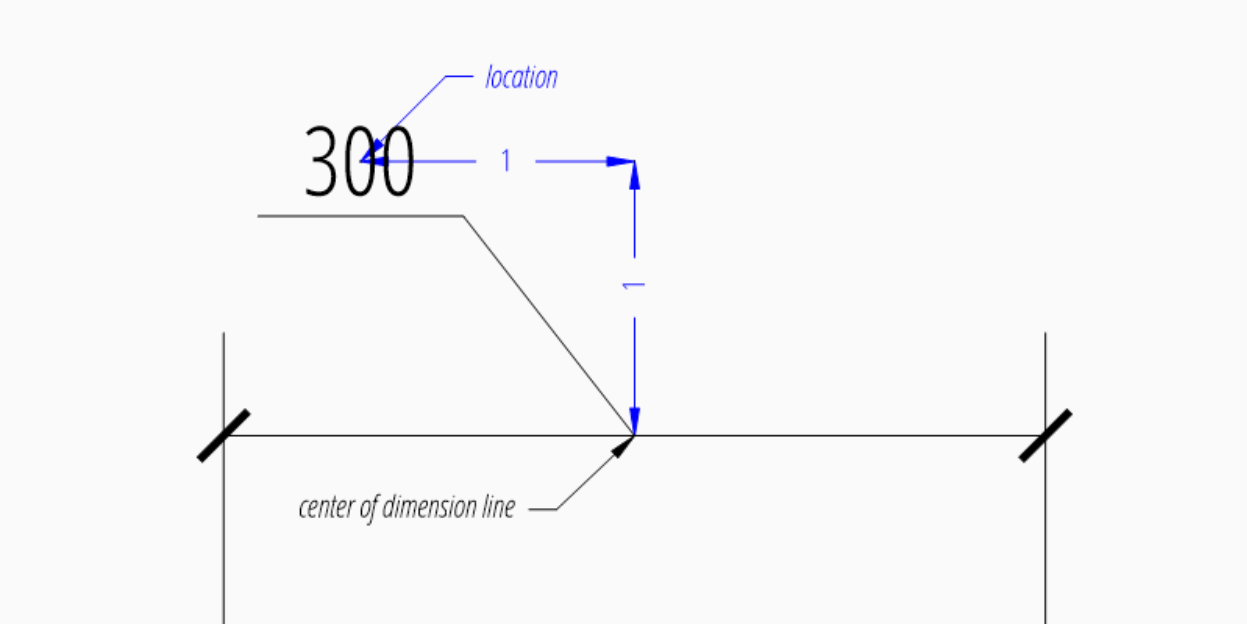


The *location* is relative to the origin of the active coordinate system or WCS if no UCS is defined in the *render()* method, the user defined *location* can also be set by *user_location_override()*.

Location Relative to Center of Dimension Line

The method *set_location()* has additional features for linear dimensions. Argument *leader* = `True` adds a simple leader from the measurement text to the center of the dimension line and argument *relative* = `True` places the measurement text relative to the center of the dimension line.

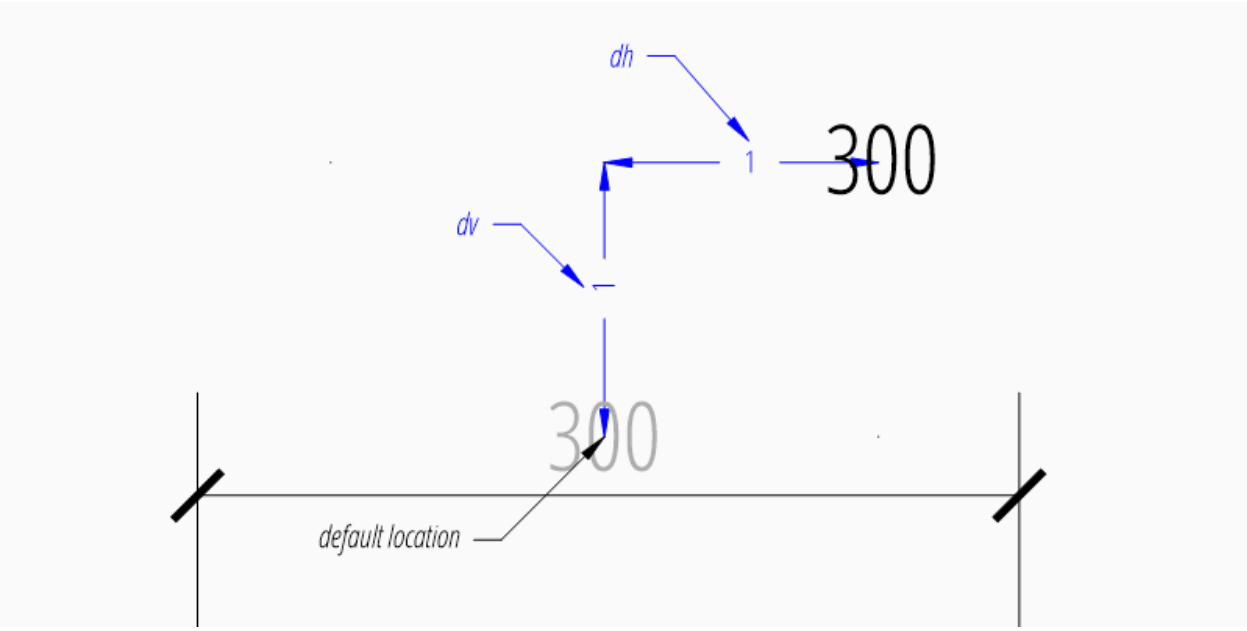
```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_location(location=(-1, 1), leader=True, relative=True)
dim.render()
```



Location Relative to Default Location

The method `shift_text()` shifts the measurement text away from the default text location. The shifting directions are aligned to the text direction, which is the direction of the dimension line in most cases, *dh* (for delta horizontal) shifts the text parallel to the text direction, *dv* (for delta vertical) shifts the text perpendicular to the text direction. This method does not support leaders.

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.shift_text(dh=1, dv=1)
dim.render()
```



Overriding Text Rotation

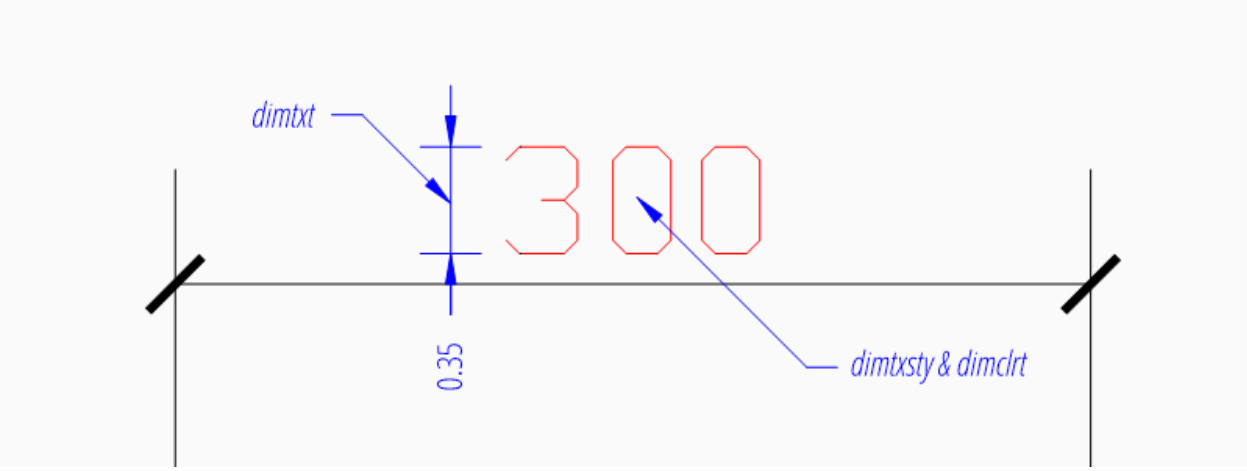
All factory methods supporting the argument *text_rotation* can override the measurement text rotation. The user defined rotation is relative to the render UCS x-axis (default is WCS).

Measurement Text Formatting and Styling

Text Properties

DIMVAR	Description
`dimtxsty`	Specifies the text style of the dimension as *`Textstyle`* name.
`dimtxt`	Text height in drawing units.
`dimclrt`	Measurement text color as *AutoCAD Color Index (ACI)*.

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimtxsty": "Standard",
        "dimtxt": 0.35,
        "dimclrt": 1,
    }
).render()
```



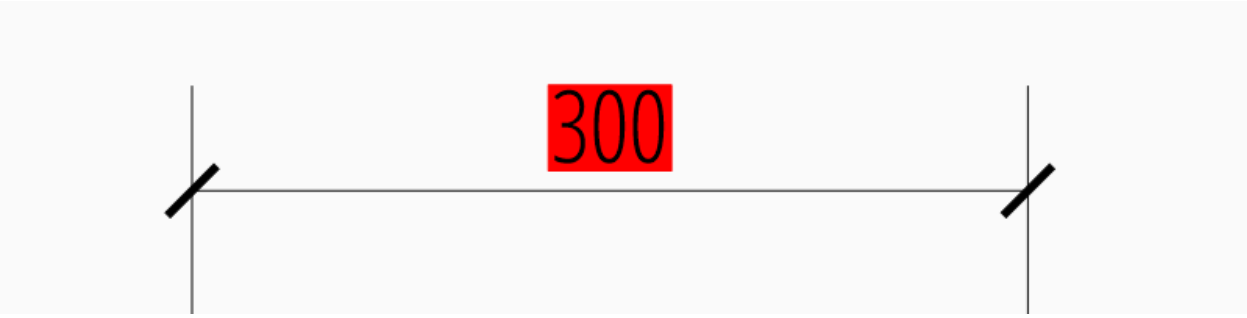
Background Filling

Background fillings are supported since DXF R2007, and *ezdxf* uses the MTEXT entity to implement this feature, so setting background filling in DXF R12 has no effect. The DIMVAR *`dimtfill`* defines the kind of background filling and the DIMVAR *`dimtfillclr`* defines the fill color.

DIMVAR	Description
`dimtfill`	Enables background filling if bigger than 0
`dimtfillclr`	Fill color as *AutoCAD Color Index (ACI)*, if `dimtfill` is 2

dimtfill	Description
0	disabled
1	canvas color
2	color defined by dimtfillclr

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimtfill": 2,
        "dimtfillclr": 1,
    }
).render()
```



Text Formatting

- **decimal places**: *dimdec* defines the number of decimal places displayed for the primary units of a dimension. (DXF R2000)
- **decimal point character**: *dimdsep* defines the decimal point as ASCII code, get the ASCII code by `ord('.')`
- **rounding**: *dimrnd*, rounds all dimensioning distances to the specified value, for instance, if `dimrnd` is set to 0.25, all distances round to the nearest 0.25 unit. If `dimrnd` is set to 1.0, all distances round to the nearest integer. For more information look at the documentation of the *ezdxf.math.xround()* function.
- **zero trimming**: *dimzin*, *ezdxf* supports only a subset of values:
  - 4 to suppress leading zeros
  - 8 to suppress trailing zeros
  - 12 as the combination of both
- **measurement factor**: scale measurement by factor *dimlfac*, e.g. to get the dimensioning text in cm for a DXF file where 1 drawing unit represents 1m, set `dimlfac` to 100.
- **text template**: *dimpost*, "<>" represents the measurement text, e.g. "~<>cm" produces "~300cm" for measurement in previous example.

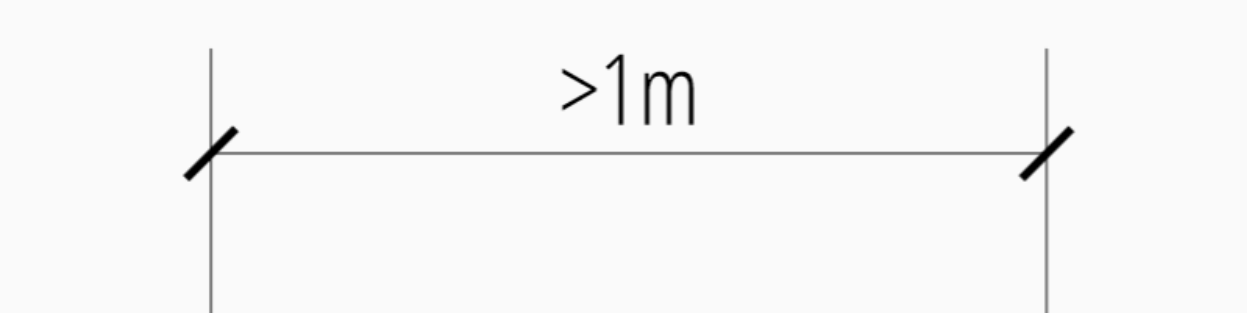
To set this values the *ezdxf.entities.DimStyle.set_text_format()* and *ezdxf.entities.DimStyleOverride.set_text_format()* methods are very recommended.

Overriding Measurement Text

This feature allows overriding the real measurement text by a custom measurement text, the text is stored as string in the *Dimension* entity as attribute *text*. Special values of the `text` attribute are: one space " " to suppress the measurement text at all, an empty string "" or "<>" to display the real measurement.

All factory functions have an explicit *text* argument, which always replaces the *text* value in the *dxfattribs* dict.

```
msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0), text=">1m").render()
```

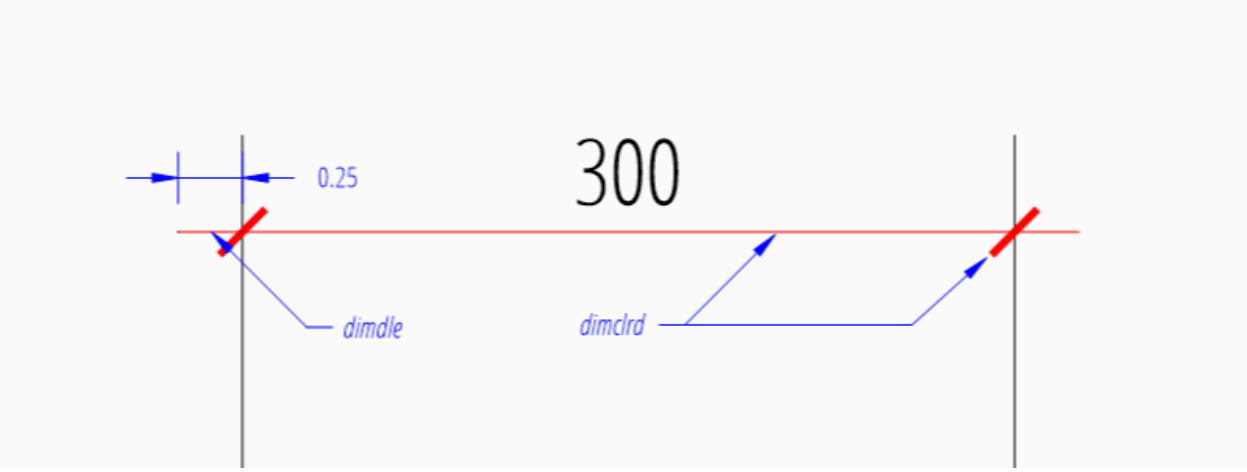


Dimension Line Properties

The *dimension line color* is defined by the DIMVAR `dimclrd` as *AutoCAD Color Index (ACI)*, `dimclrd` and also defines the color of the arrows. The *linetype* is defined by `dimltype` and requires DXF R2007. The *lineweight* is defined by `dimlwd` and requires DXF R2000, see also the *lineweight* reference for valid values. The `dimdle` is the extension of the dimension line beyond the extension lines, this dimension line extension is not supported for all arrows.

DIMVAR	Description
`dimclrd`	dimension line and arrows color as *AutoCAD Color Index (ACI)*
`dimltype`	linetype of dimension line
`dimlwd`	line weight of dimension line
`dimdle`	extension of dimension line in drawing units

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimclrd": 1,  # red
        "dimdle": 0.25,
        "dimltype": "DASHED2",
        "dimlwd": 35,  # 0.35mm line weight
    }
).render()
```



DimStyleOverride() method:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_dimline_format(
    color=1, linetype="DASHED2", lineweight=35, extension=0.25
)
dim.render()
```

Extension Line Properties

The *extension line color* is defined by the DIMVAR `dimclre` as *AutoCAD Color Index (ACI)*. The *linetype* for the first and the second extension line is defined by `dimltex1` and `dimltex2` and requires DXF R2007. The *lineweight* is defined by `dimlwe` and required DXF R2000, see also the *`lineweight`* reference for valid values.

The `dimexe` is the extension of the extension line beyond the dimension line, and `dimexo` defines the offset of the extension line from the measurement point.

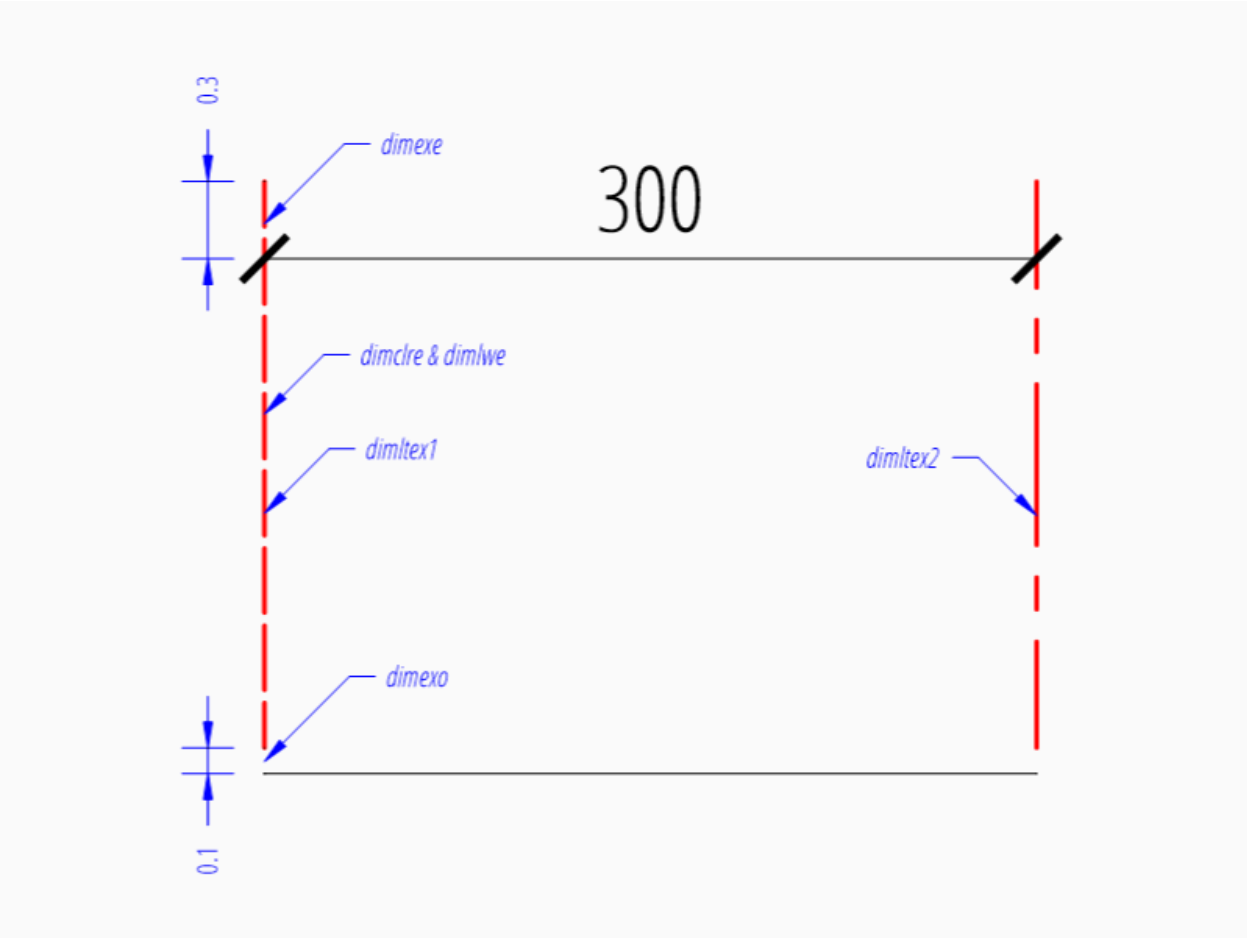
DIMVAR	Description
`dimclre`	extension line color as *AutoCAD Color Index (ACI)*
`dimltex1`	linetype of first extension line
`dimltex2`	linetype of second extension line
`dimlwe`	line weight of extension line
`dimexe`	extension beyond dimension line in drawing units
`dimexo`	offset of extension line from measurement point
`dimfxlon`	set to 1 to enable fixed length extension line
`dimfxl`	length of fixed length extension line in drawing units
`dimse1`	suppress first extension line if 1
`dimse2`	suppress second extension line if 1

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimclre": 1,   # red
        "dimltex1": "DASHED2",
        "dimltex2": "CENTER2",
```

(continues on next page)

(continued from previous page)

```
        "dimlwe": 35,   # 0.35mm line weight
        "dimexe": 0.3,  # length above dimension line
        "dimexo": 0.1,  # offset from measurement point
    }
).render()
```



DimStyleOverride() methods:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_extline_format(color=1, lineweight=35, extension=0.3, offset=0.1)
dim.set_extline1(linetype="DASHED2")
dim.set_extline2(linetype="CENTER2")
dim.render()
```

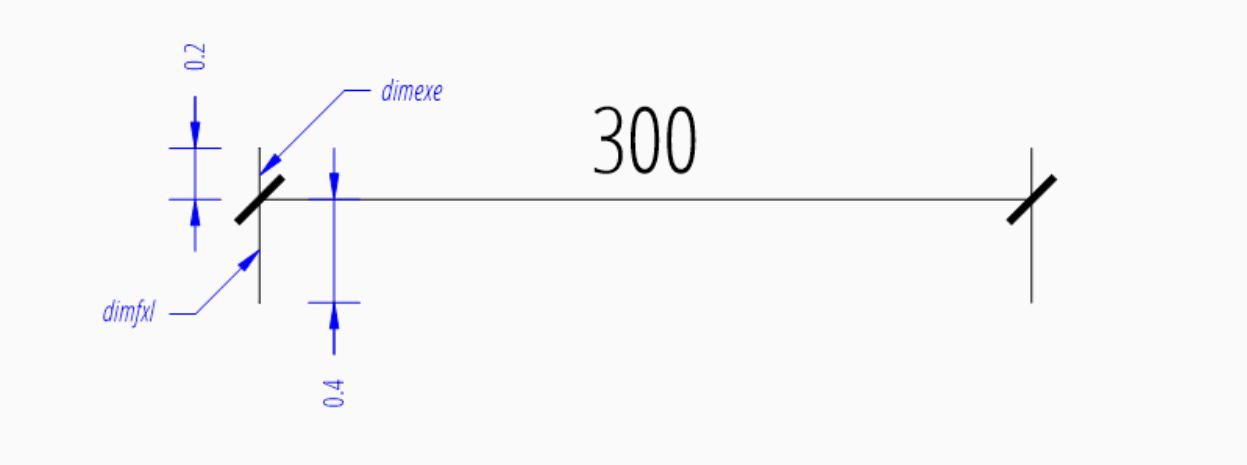
Fixed length extension lines are supported in DXF R2007, set `dimfxlon` to 1 and `dimfxl` defines the length of the extension line starting at the dimension line.

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimfxlon": 1,  # fixed length extension lines
        "dimexe": 0.2,  # length above dimension line
```

(continues on next page)

(continued from previous page)

```
        "dimfxl": 0.4,  # length below dimension line
    }
).render()
```



DimStyleOverride() method:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_extline_format(extension=0.2, fixed_length=0.4)
dim.render()
```

To suppress extension lines set `dimse1` to 1 to suppress the first extension line and `dimse2` to 1 to suppress the second extension line.

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimse1": 1,  # suppress first extension line
        "dimse2": 1,  # suppress second extension line
        "dimblk": ezdxf.ARROWS.closed_filled,  # arrows just looks better
    }
).render()
```

DimStyleOverride() methods:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_arrows(blk=ezdxf.ARROWS.closed_filled)
dim.set_extline1(disable=True)
```

(continues on next page)

(continued from previous page)

```
dim.set_extline2(disable=True)
dim.render()
```

Arrows

"Arrows" mark then beginning and the end of a dimension line, and most of them do not look like arrows.

DXF distinguish between the simple tick (a slanted line) and arrows as blocks.

To use a simple tick as "arrow" set *dimtsz* to a value greater than 0, this also disables arrow blocks as side effect:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_tick(size=0.25)
dim.render()
```

Ezdxf uses the "ARCHTICK" block at double size to render the tick (AutoCAD and BricsCad just draw a simple line), so there is no advantage of using the tick instead of an arrow.

Using arrows:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_arrow(blk="OPEN_30", size=0.25)
dim.render()
```

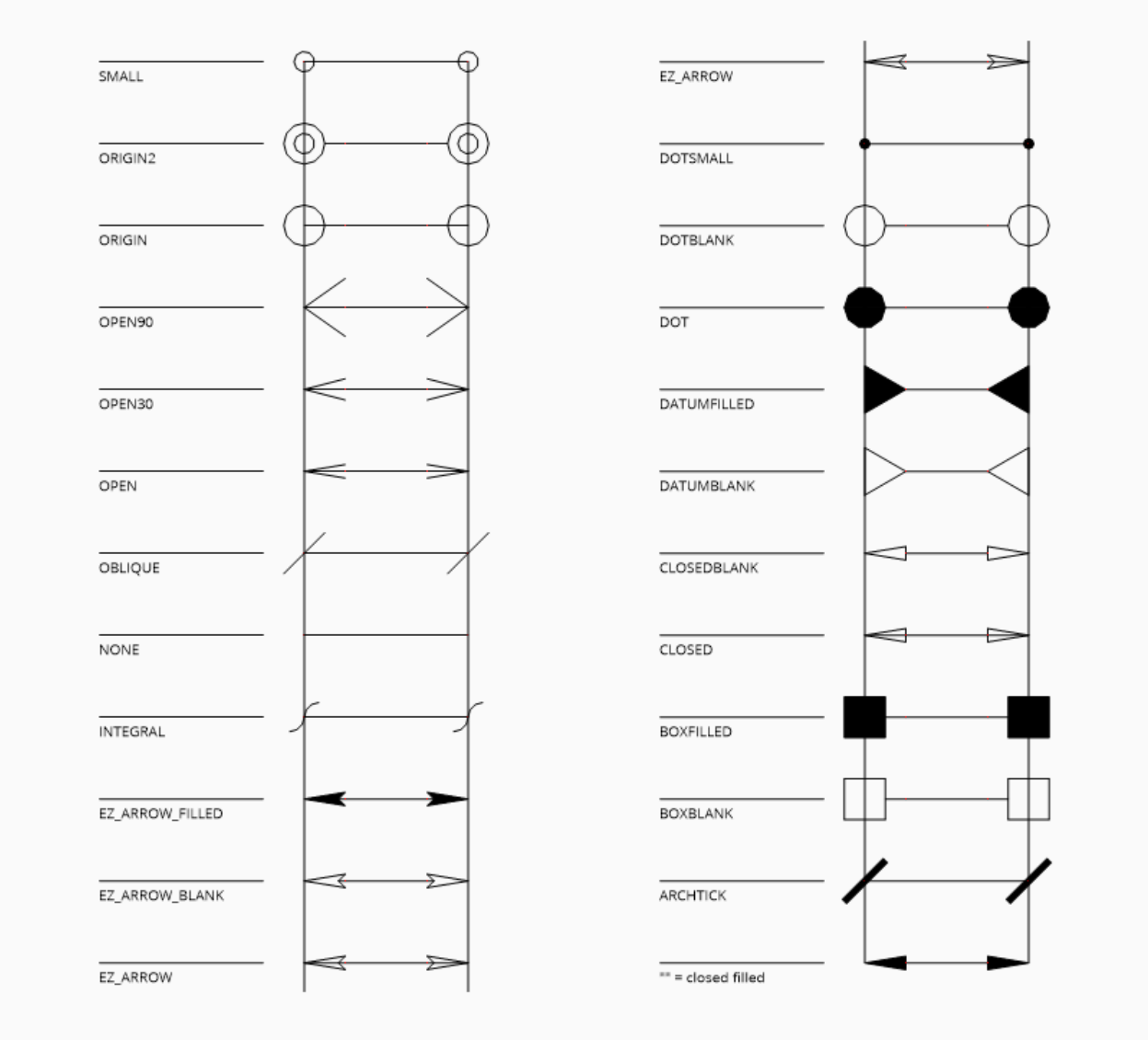
DIMVAR	Description
dimtsz	tick size in drawing units, set to 0 to use arrows
dimblk	set both arrow block names at once
dimblk1	first arrow block name
dimblk2	second arrow block name
dimasz	arrow size in drawing units

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimtsz": 0,  # set tick size to 0 to enable arrow usage
        "dimasz": 0.25,  # arrow size in drawing units
        "dimblk": "OPEN_30",  # arrow block name
    }
).render()
```

The dimension line extension (dimdle) works only for a few arrow blocks and the simple tick:

- "ARCHTICK"
- "OBLIQUE"
- "NONE"
- "SMALL"
- "DOTSMALL"
- "INTEGRAL"

Arrow Shapes



Arrow Names

The arrow names are stored as attributes in the `ezdxf.ARROWS` object.

closed_filled	"" (empty string)
dot	"DOT"
dot_small	"DOTSMALL"
dot_blank	"DOTBLANK"
origin_indicator	"ORIGIN"
origin_indicator_2	"ORIGIN2"
open	"OPEN"
right_angle	"OPEN90"
open_30	"OPEN30"
closed	"CLOSED"
dot_smallblank	"SMALL"
none	"NONE"
oblique	"OBLIQUE"
box_filled	"BOXFILLED"
box	"BOXBLANK"
closed_blank	"CLOSEDBLANK"
datum_triangle_filled	"DATUMFILLED"
datum_triangle	"DATUMBLANK"
integral	"INTEGRAL"
architectural_tick	"ARCHTICK"
ez_arrow	"EZ_ARROW"
ez_arrow_blank	"EZ_ARROW_BLANK"
ez_arrow_filled	"EZ_ARROW_FILLED"

Tolerances and Limits

The tolerances and limits features are implemented by using inline codes for the *MText* entity, therefore DXF R2000 is required. It is not possible to use both tolerances and limits at the same time.

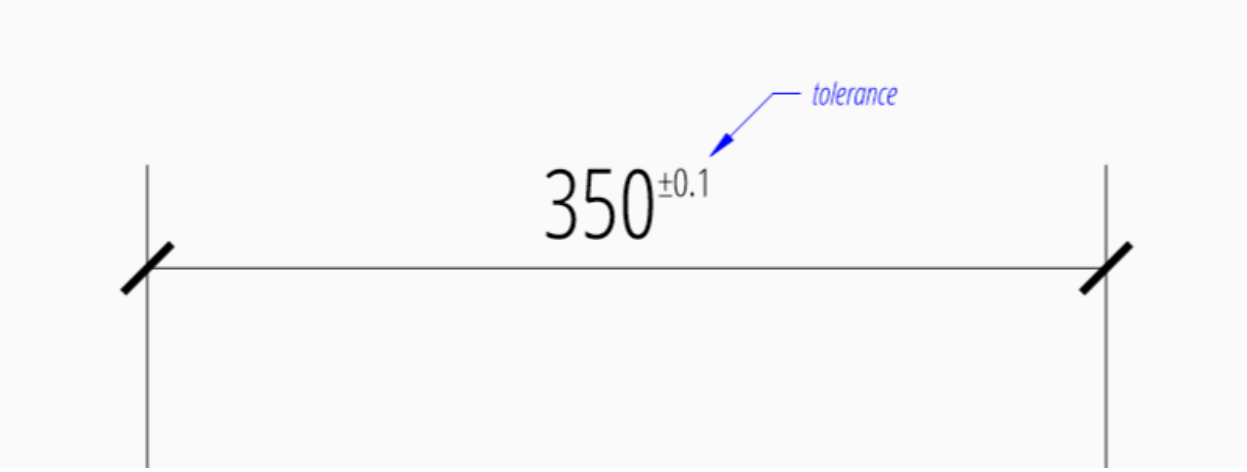
Tolerances

Geometrical tolerances are shown as additional text appended to the measurement text. It is recommend to use *set_tolerance()* method in *DimStyleOverride* or *DimStyle*.

The attribute `dimtp` defines the upper tolerance value, `dimtm` defines the lower tolerance value if present, else the lower tolerance value is the same as the upper tolerance value. Tolerance values are shown as given!

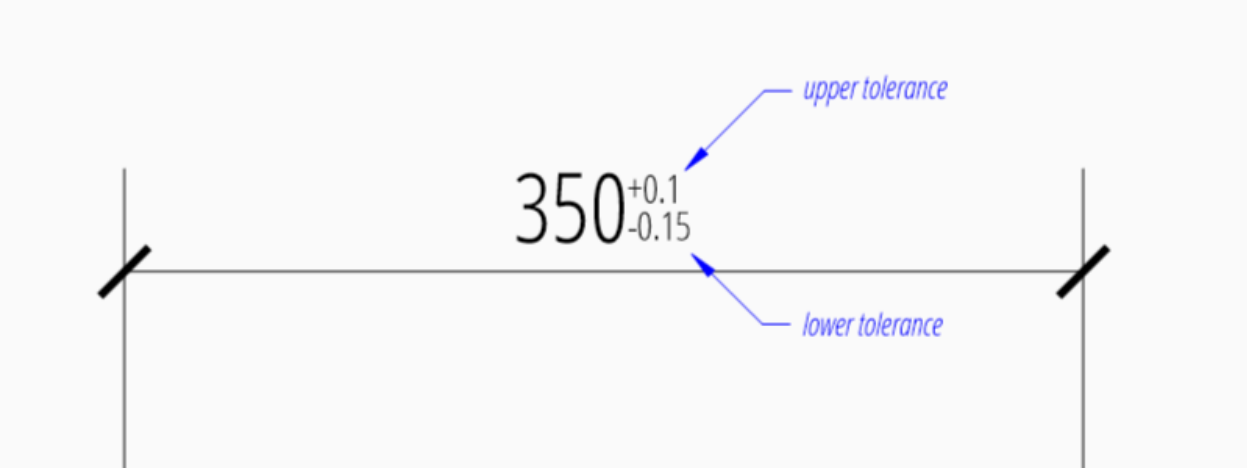
Same upper and lower tolerance value:

```
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_tolerance(.1, hfactor=.4, align="top", dec=2)
dim.render()
```



Different upper and lower tolerance values:

```
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_tolerance(upper=.1, lower=.15, hfactor=.4, align="middle", dec=2)
dim.render()
```



The attribute `dimtfac` specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by `dimtxt`. For example, if `dimtfac` is set to 1.0, the text height of fractions and tolerances is the same height as the dimension text. If `dimtxt` is set to 0.75, the text height of limits and tolerances is three-quarters the size of dimension text.

Vertical justification for tolerances is specified by `dimtolj`:

`dimtolj`	Description
0	Align with bottom line of dimension text
1	Align vertical centered to dimension text
2	Align with top line of dimension text

DIMVAR	Description
dimtol	set to 1 to enable tolerances
dimtp	set the maximum (or upper) tolerance limit for dimension text
dimtm	set the minimum (or lower) tolerance limit for dimension text
dimtfac	specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by dimtxt.
dimtzin	4 to suppress leading zeros, 8 to suppress trailing zeros or 12 to suppress both, like dimzin for dimension text, see also *Text Formatting*
dimtolj	set the vertical justification for tolerance values relative to the nominal dimension text.
dimtdec	set the number of decimal places to display in tolerance values

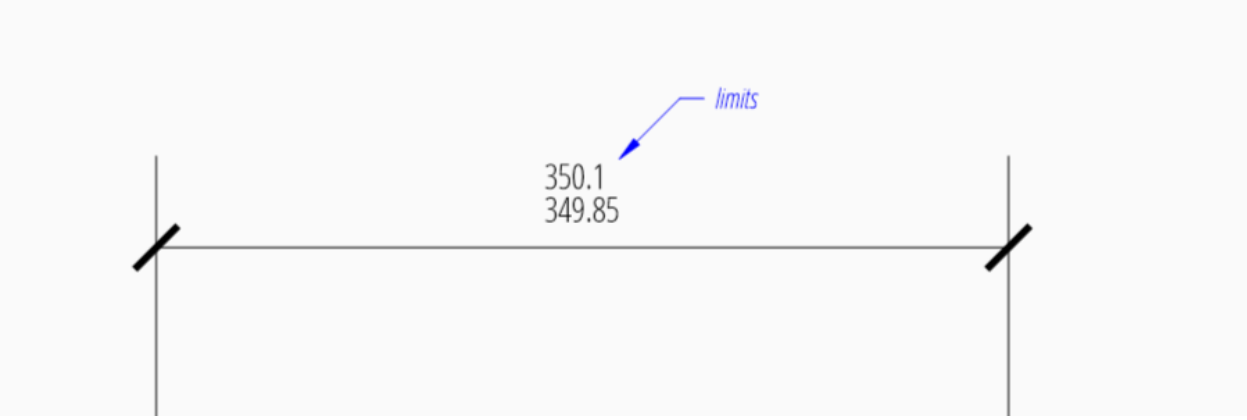
Limits

The geometrical limits are shown as upper and lower measurement limit and replaces the usual measurement text. It is recommend to use *set_limits()* method in *DimStyleOverride* or *DimStyle*.

For limits the tolerance values are drawing units scaled by measurement factor dimlfac, the upper limit is scaled measurement value + dimtp and the lower limit is scaled measurement value - dimtm.

The attributes dimtfac, dimtzin and dimtdec have the same meaning for limits as for tolerances.

```
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_limits(upper=.1, lower=.15, hfactor=.4, dec=2)
dim.render()
```



DIMVAR	Description
dimlim	set to 1 to enable limits

Alternative Units

Alternative units are not supported.

6.5.24 Tutorial for Radius Dimensions

Please read the *Tutorial for Linear Dimensions* before, if you haven't.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

```
import ezdxf

# DXF R2010 drawing, official DXF version name: 'AC1024',
# setup=True setups the default dimension styles
doc = ezdxf.new("R2010", setup=True)

msp = doc.modelspace()  # add new dimension entities to the modelspace
msp.add_circle((0, 0), radius=3)  # add a CIRCLE entity, not required
# add default radius dimension, measurement text is located outside
dim = msp.add_radius_dim(
    center=(0, 0), radius=3, angle=45, dimstyle="EZ_RADIUS"
)
# necessary second step, to create the BLOCK entity with the dimension geometry.
dim.render()
doc.saveas("radius_dimension.dxf")
```

The example above creates a 45 degrees slanted radius `Dimension` entity, the default dimension style "EZ_RADIUS" is defined as 1 drawing unit = 1m, drawing scale = 1:100 and the length factor = 100, which creates a measurement text in cm, the default location for the measurement text is outside of the circle.

The *center* point defines the center of the circle but there doesn't have to exist a circle entity, *radius* defines the circle radius, which is also the measurement, and angle defines the slope of the dimension line, it is also possible to define the circle by a measurement point *mpoint* on the circle.

The return value *dim* is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as `dim.dimension`.

Placing Measurement Text

There are different predefined DIMSTYLES to achieve various text placing locations.

The basic DIMSTYLE "EZ_RADIUS" settings are:

- 1 drawing unit = 1m
- scale 1:100
- the length factor `dimlfac` = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size `dimasz` = 0.25

Note: Not all possibles features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the radial dimension there are less features implemented than for the linear dimension because of the lack of good documentation.

See also:

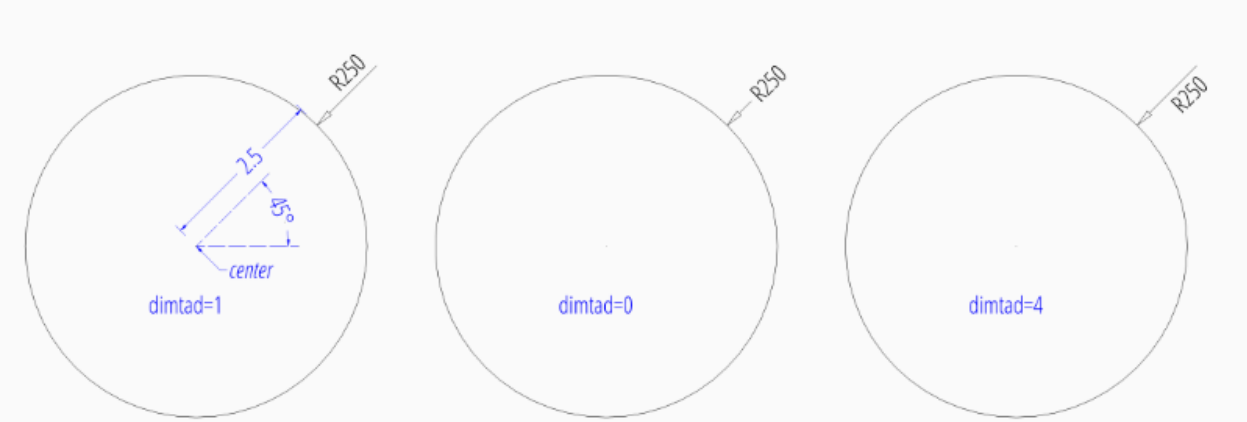
- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file standards.py shows how to create your own DIMSTYLES.
- The Script dimension_radius.py shows examples for radius dimensions.

Default Text Locations Outside

Advanced "EZ_RADIUS" settings for placing the text outside of the circle:

tmove	1 = add a leader when dimension text is moved, this is the best setting for text outside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dimtad	1 = place the text vertical above the dimension line

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS"
)
dim.render()  # always required, but not shown in the following examples
```



To force text outside horizontal set *dimtoh* to 1:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```

Default Text Locations Inside

DIMSTYLE "EZ_RADIUS_INSIDE" can be used to place the dimension text inside the circle at a default location.

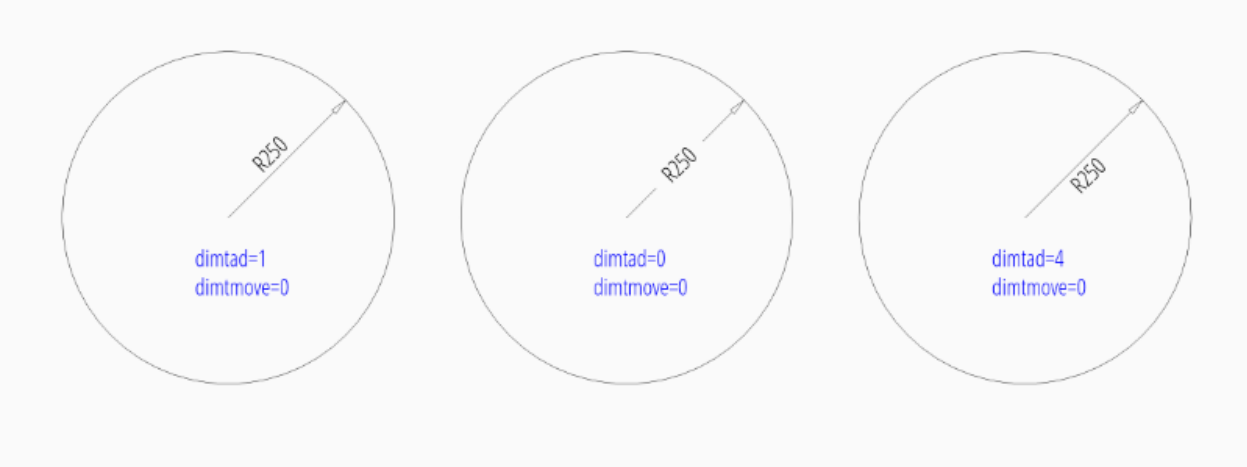
The basic DIMSTYLE "EZ_RADIUS_INSIDE" settings are:

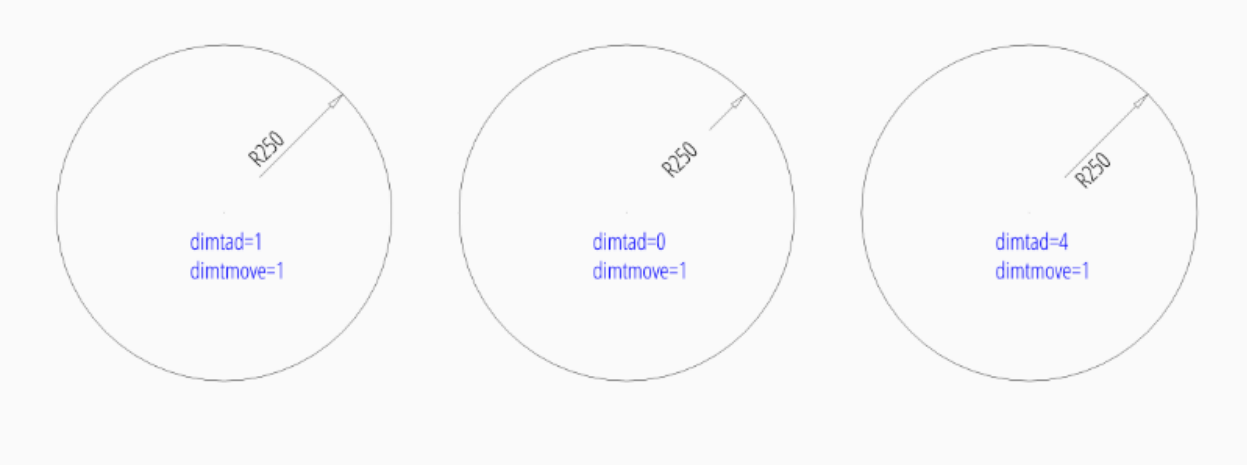
- 1 drawing unit = 1m
- scale 1:100, length_factor is 100 which creates
- the length factor `dimlfac` = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size `dimasz` = 0.25

Advanced "EZ_RADIUS_INSIDE" settings to place (force) the text inside of the circle:

tmov	0 = moves the dimension line with dimension text, this is the best setting for text inside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dimti	1 = force text inside
di-mat-fit	0 = force text inside, required by BricsCAD and AutoCAD
dim-tad	0 = center text vertical, BricsCAD and AutoCAD always create a vertical centered text, *ezdxf* let you choose the vertical placement (above, below, center), but editing the DIMENSION in BricsCAD or AutoCAD will reset text to center placement.

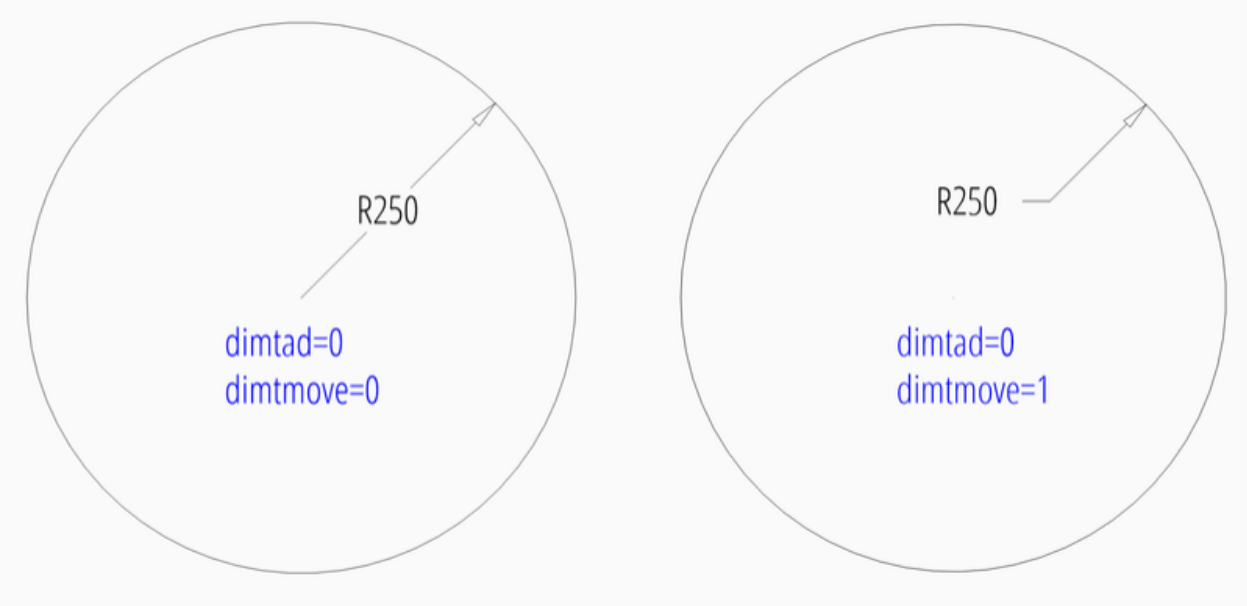
```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS_INSIDE"
)
```





To force text inside horizontal set *dimtih* to 1:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS_INSIDE",
    override={"dimtih": 1}
)
```

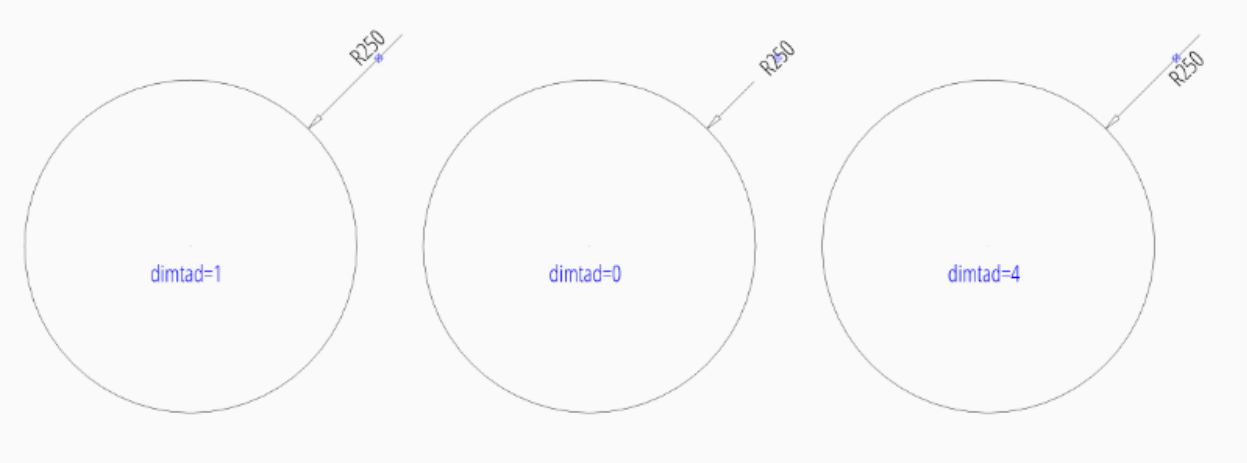


User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location. This location also determines the angle of the dimension line and overrides the argument *angle*. For user defined locations it is not necessary to force text inside (`dimtix=1`), because the location of the text is explicit given, therefore the DIMSTYLE "EZ_RADIUS" can be used for all this examples.

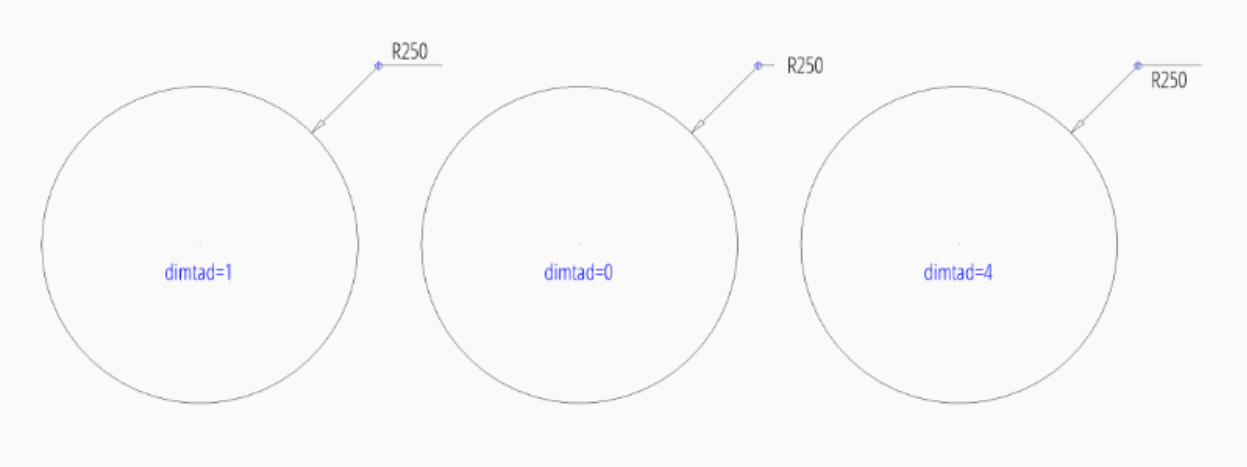
User defined location outside of the circle:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    location=(4, 4),
    dimstyle="EZ_RADIUS"
)
```



User defined location outside of the circle and forced horizontal text:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    location=(4, 4),
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```

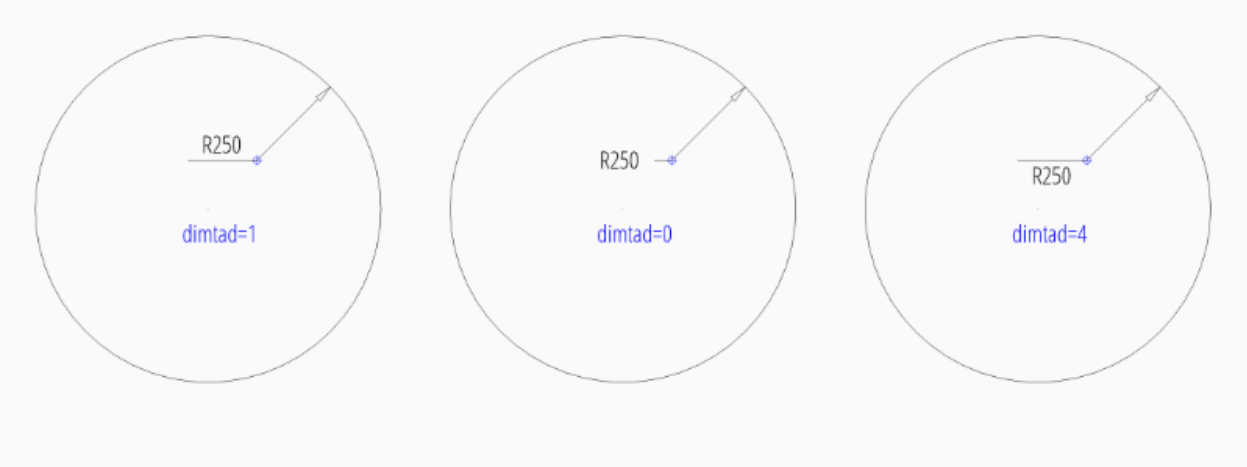


User defined location inside of the circle:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    location=(1, 1),
    dimstyle="EZ_RADIUS"
)
```

User defined location inside of the circle and forced horizontal text:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    location=(1, 1),
    dimstyle="EZ_RADIUS",
    override={"dimtih": 1},
)
```



Center Mark/Lines

Center mark/lines are controlled by *dimcen*, default value is 0 for predefined dimstyles "EZ_RADIUS" and "EZ_RADIUS_INSIDE":

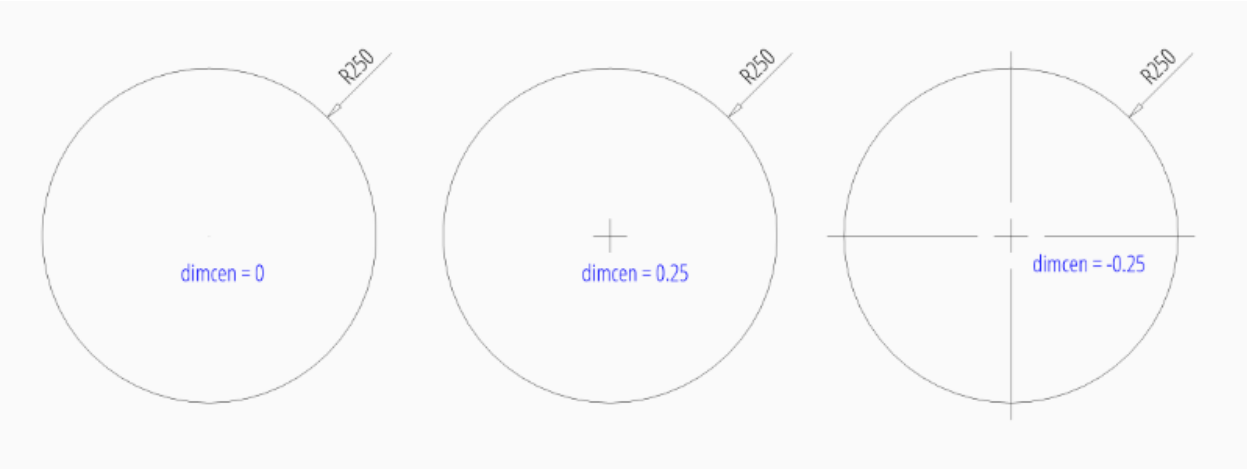
0	Center mark is off
>0	Create center mark of given size
<0	Create center lines

```
dim = msp.add_radius_dim(
    center=(0, 0),
```

(continues on next page)

(continued from previous page)

```
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS",
    override={"dimcen": 0.25},
)
```



Overriding Measurement Text

See Linear Dimension Tutorial: *Overriding Measurement Text*

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: *Measurement Text Formatting and Styling*

6.5.25 Tutorial for Diameter Dimensions

Please read the *Tutorial for Radius Dimensions* before, if you haven't.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

This is a repetition of the radius tutorial, just with diameter dimensions.

```
import ezdxf

# setup=True setups the default dimension styles
doc = ezdxf.new("R2010", setup=True)

msp = doc.modelspace()  # add new dimension entities to the modelspace
msp.add_circle((0, 0), radius=3)  # add a CIRCLE entity, not required
# add default diameter dimension, measurement text is located outside
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=3,
    angle=45,
```

(continues on next page)

(continued from previous page)

```
    dimstyle="EZ_RADIUS"
)
dim.render()
doc.saveas("diameter_dimension.dxf")
```

The example above creates a 45 degrees slanted diameter `Dimension` entity, the default dimension style "EZ_RADIUS" (same as for radius dimensions) is defined as 1 drawing unit = 1m, drawing scale = 1:100 and the length factor = 100, which creates a measurement text in cm, the default location for the measurement text is outside of the circle.

The *center* point defines the center of the circle but there doesn't have to exist a circle entity, *radius* defines the circle radius and *angle* defines the slope of the dimension line, it is also possible to define the circle by a measurement point *mpoint* on the circle.

The return value *dim* is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as *dim.dimension*.

Placing Measurement Text

There are different predefined DIMSTYLES to achieve various text placing locations.

The basic DIMSTYLE "EZ_RADIUS" settings are:

- 1 drawing unit = 1m
- scale 1:100
- the length factor `dimlfac` = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size `dimasz` = 0.25

Note: Not all possibles features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the diameter dimension there are less features implemented than for the linear dimension because of the lack of good documentation.

See also:

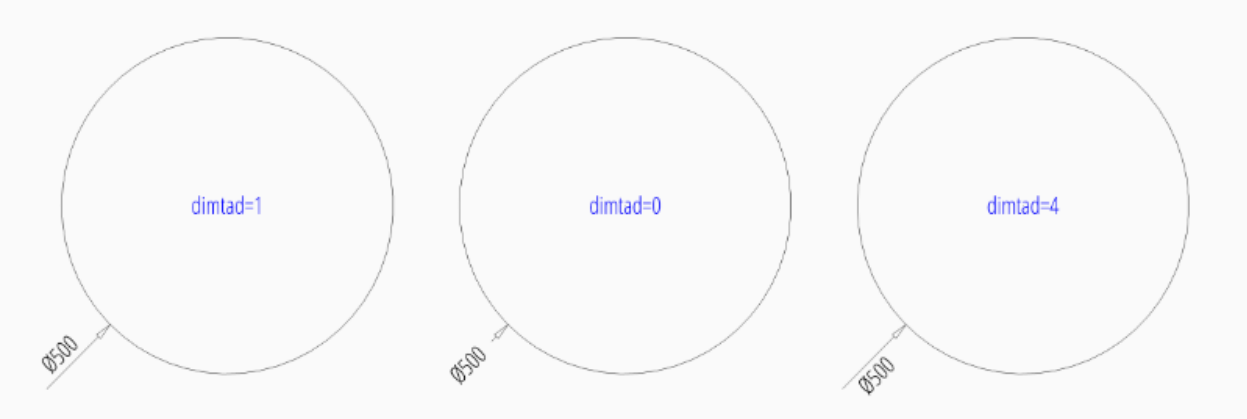
- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file standards.py shows how to create your own DIMSTYLES.
- The Script dimension_diameter.py shows examples for radius dimensions.

Default Text Locations Outside

"EZ_RADIUS" default settings for to place text outside:

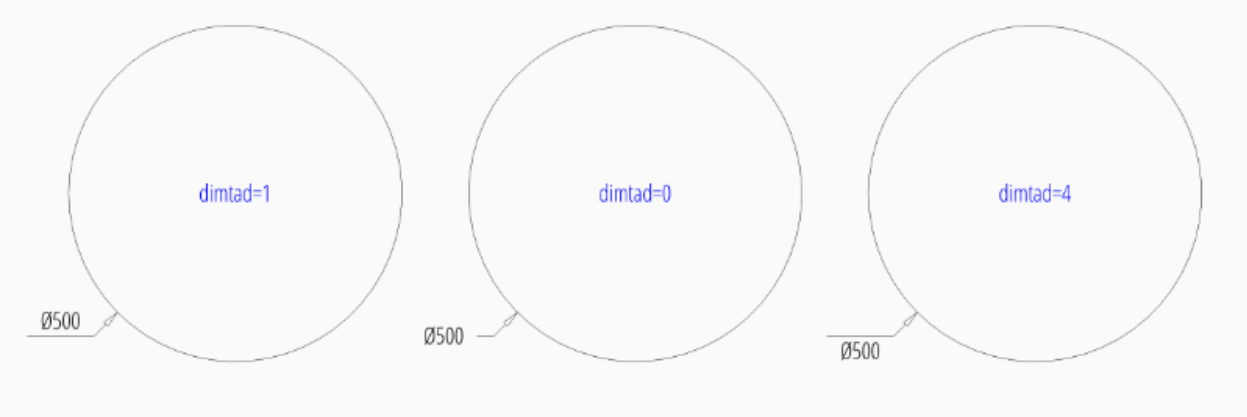
tmove	1 = add a leader when dimension text is moved, this is the best setting for text outside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dimtad	1 = place the text vertical above the dimension line

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS"
)
dim.render()  # always required, but not shown in the following examples
```



To force text outside horizontal set *dimtoh* to 1:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```



Default Text Locations Inside

DIMSTYLE "EZ_RADIUS_INSIDE" can be used to place the dimension text inside the circle at a default location.

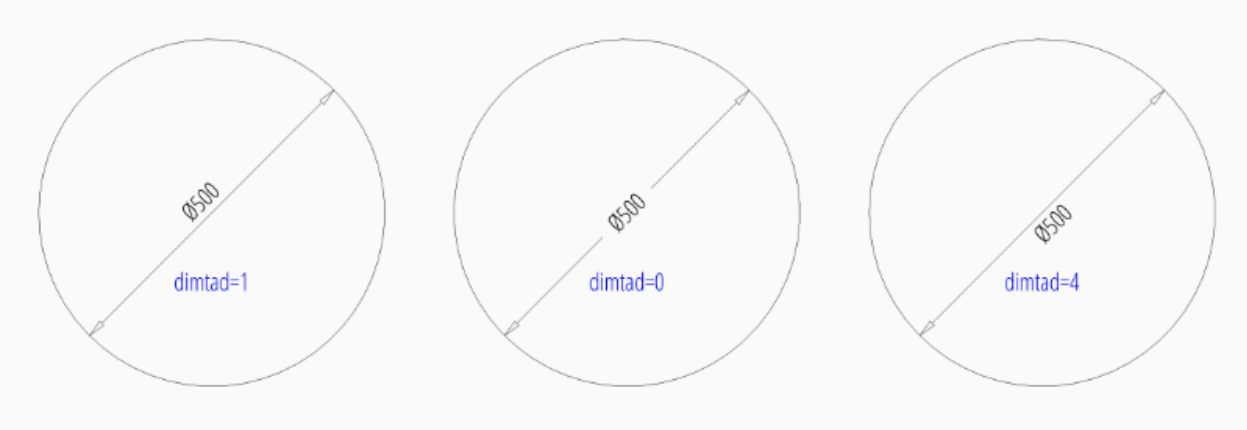
The basic DIMSTYLE settings are:

- 1 drawing unit = 1m
- scale 1:100, length_factor is 100 which creates
- the length factor `dimlfac` = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size `dimasz` = 0.25

Advanced "EZ_RADIUS_INSIDE" settings to place (force) the text inside of the circle:

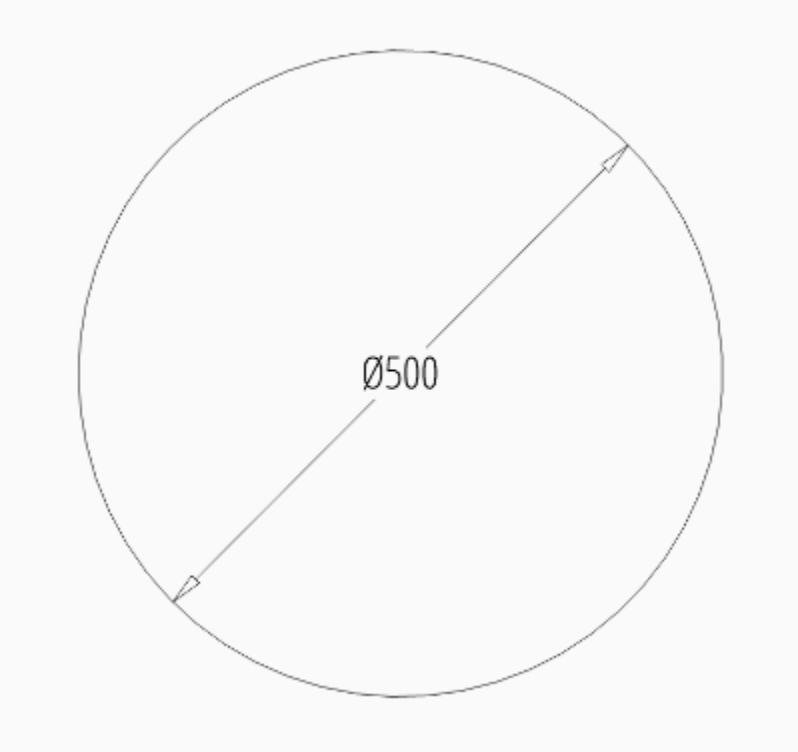
tmov	0 = moves the dimension line with dimension text, this is the best setting for text inside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dimti	1 = force text inside
di-mat-fit	0 = force text inside, required by BricsCAD and AutoCAD
dim-tad	0 = center text vertical, BricsCAD and AutoCAD always create a vertical centered text, *ezdxf* let you choose the vertical placement (above, below, center), but editing the DIMENSION in BricsCAD or AutoCAD will reset text to center placement.

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS_INSIDE"
)
```



To force text inside horizontal set *dimtih* to 1:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS_INSIDE",
    override={"dimtih": 1}
)
```

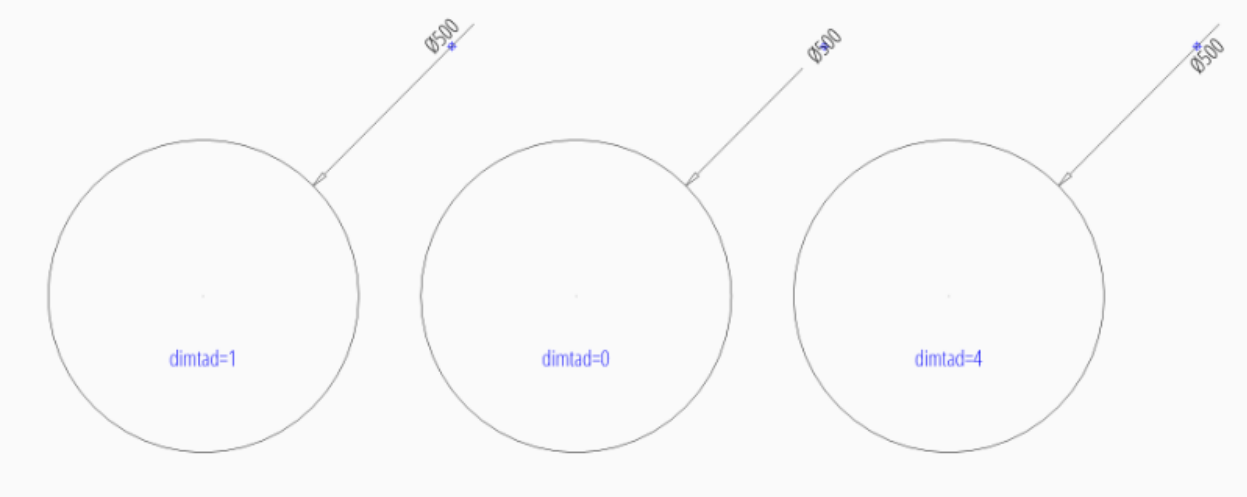


User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location. This location also determines the angle of the dimension line and overrides the argument *angle*. For user defined locations it is not necessary to force text inside (`dimtix=1`), because the location of the text is explicit given, therefore the DIMSTYLE "EZ_RADIUS" can be used for all this examples.

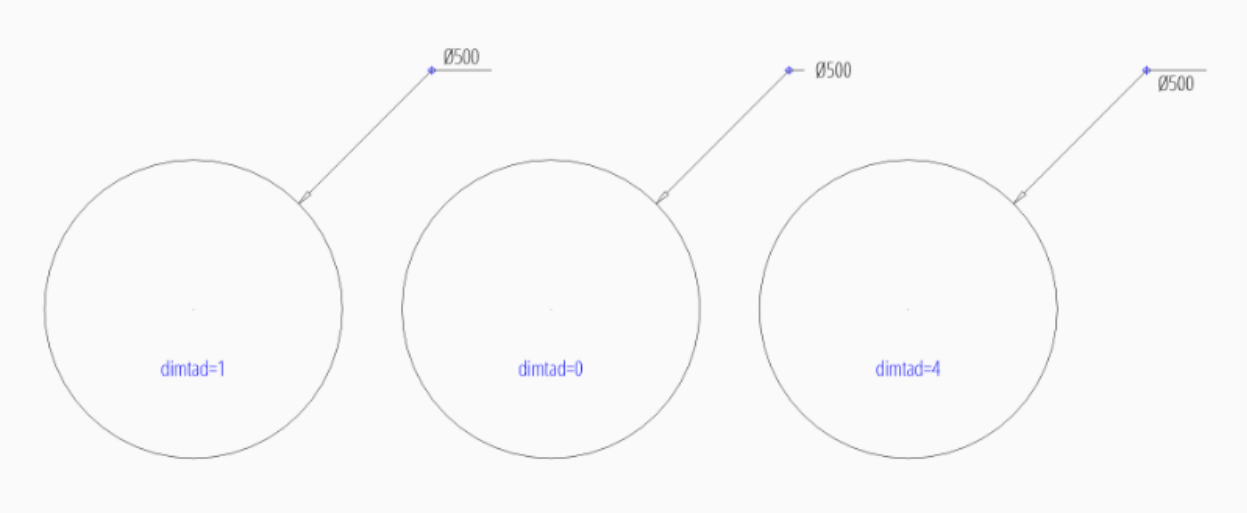
User defined location outside of the circle:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    location=(4, 4),
    dimstyle="EZ_RADIUS"
)
```



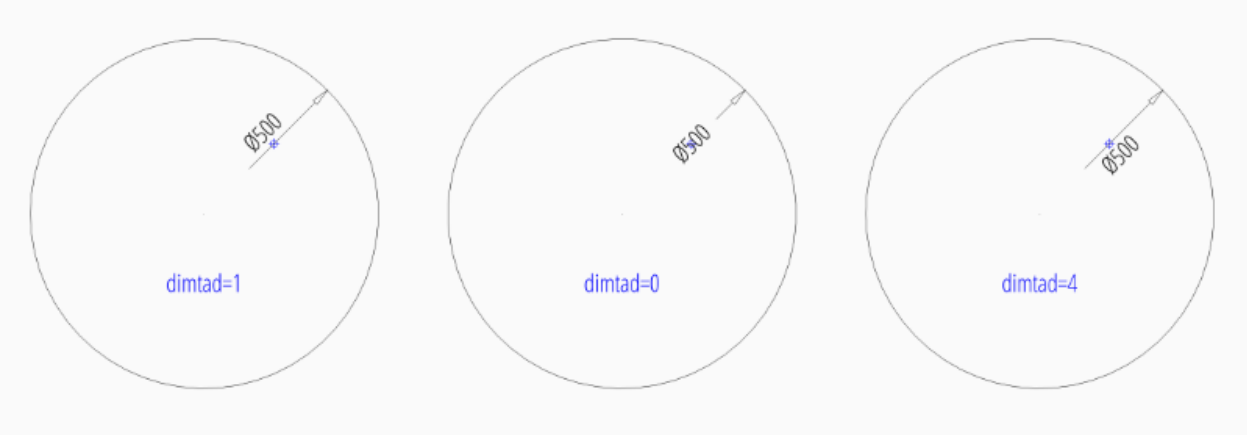
User defined location outside of the circle and forced horizontal text:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    location=(4, 4),
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```



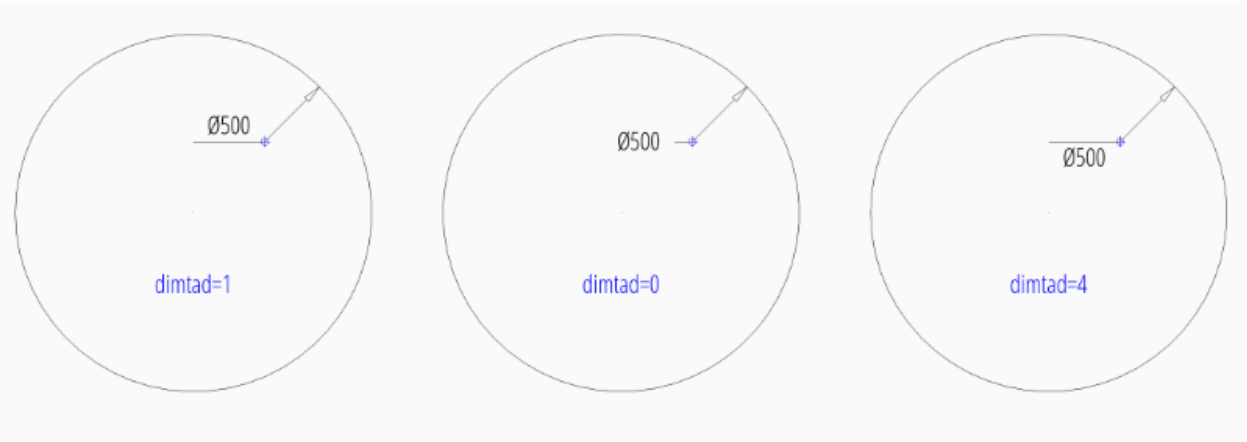
User defined location inside of the circle:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    location=(1, 1),
    dimstyle="EZ_RADIUS"
)
```



User defined location inside of the circle and forced horizontal text:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    location=(1, 1),
    dimstyle="EZ_RADIUS",
    override={"dimtih": 1},
)
```



Center Mark/Lines

See Radius Dimension Tutorial: *Center Mark/Lines*

Overriding Measurement Text

See Linear Dimension Tutorial: *Overriding Measurement Text*

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: *Measurement Text Formatting and Styling*

6.5.26 Tutorial for Angular Dimensions

Please read the *Tutorial for Linear Dimensions* before, if you haven't.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Dimension Style "EZ_CURVED"

All factory methods to create angular dimensions uses the dimension style "EZ_CURVED" for curved dimension lines which is defined as:

- angle unit is decimal degrees, `dimaunit` = 0
- measurement text height = 0.25 (drawing scale = 1:100)
- measurement text location is above the dimension line
- closed filled arrow and arrow size `dimasz` = 0.25
- `dimazin` = 2, suppresses trailing zeros (e.g. 12.5000 becomes 12.5)

This DIMENSION style only exist if the argument *setup* is `True` for creating a new DXF document by *`ezdxf.new()`*. Every dimension style which does not exist will be replaced by the dimension style "Standard" at DXF export by `save()` or `saveas()` (e.g. dimension style setup was not initiated).

Add all *ezdxf* specific resources (line types, text- and dimension styles) to an existing DXF document:

```
import ezdxf
from ezdxf.tools.standards import setup_drawing

doc = ezdxf.readfile("your.dxf")
setup_drawing(doc, topics="all")
```

Factory Methods to Create Angular Dimensions

Defined by Center, Radius and Angles

The first example shows an angular dimension defined by the center point, radius, start- and end angles:

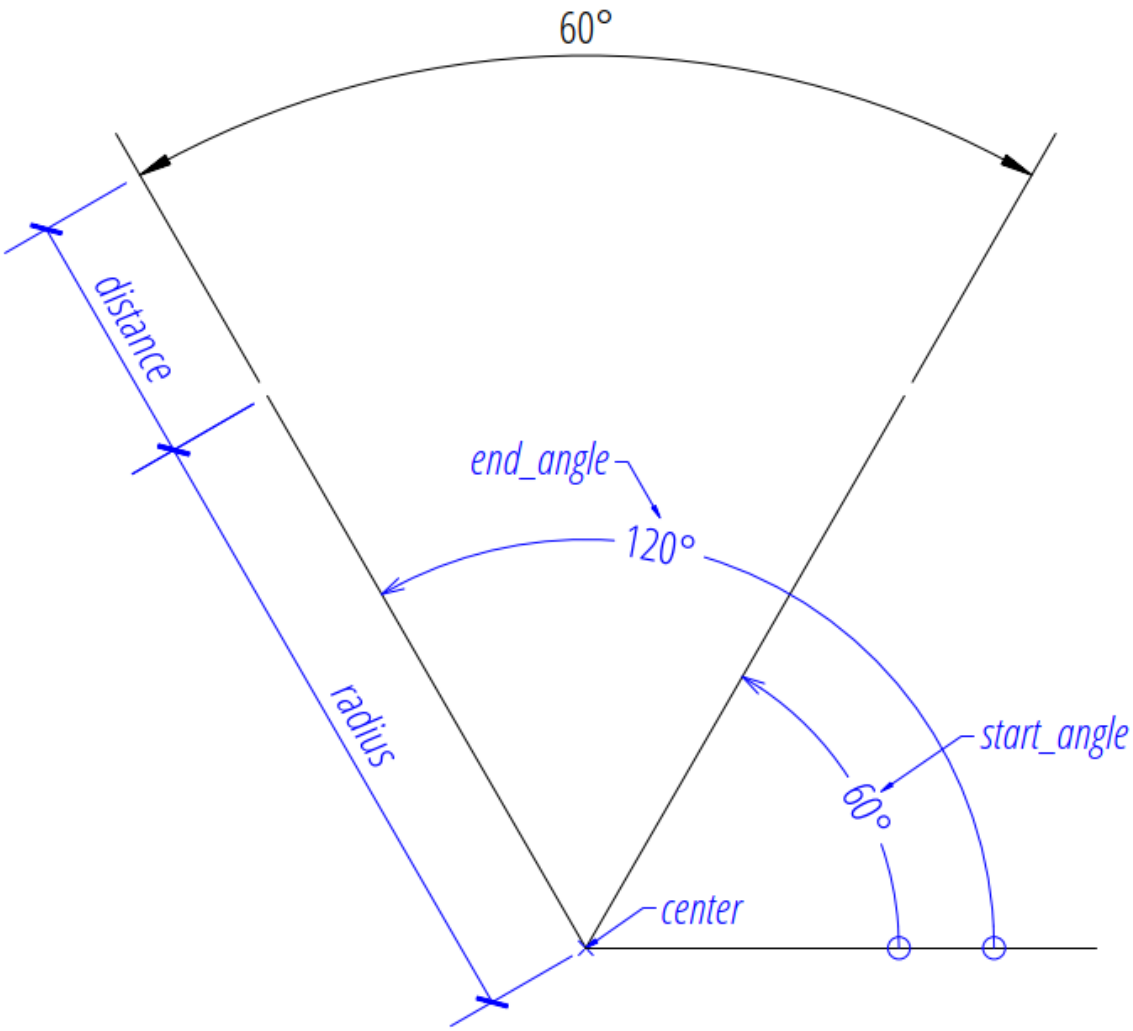
```
import ezdxf

# Create a DXF R2010 document:
# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new("R2010", setup=True)

# Add new entities to the modelspace:
msp = doc.modelspace()

# Add an angular DIMENSION defined by the center point, start- and end angles,
```

(continues on next page)

(continued from previous page)

```
# the measurement text is placed at the default location above the dimension
# line:
dim = msp.add_angular_dim_cra(
    center=(5, 5),  # center point of the angle
    radius= 7,  # distance from center point to the start of the extension lines
    start_angle=60,  # start angle in degrees
    end_angle=120,  # end angle in degrees
    distance=3,  # distance from start of the extension lines to the dimension line
    dimstyle="EZ_CURVED",  # default angular dimension style
)

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
dim.render()
doc.saveas("angular_dimension_cra.dxf")
```

The return value *dim* is **not** a dimension entity, instead a *DimStyleOverride* object is returned, the dimension entity is stored as `dim.dimension`.



Angle by 2 Lines

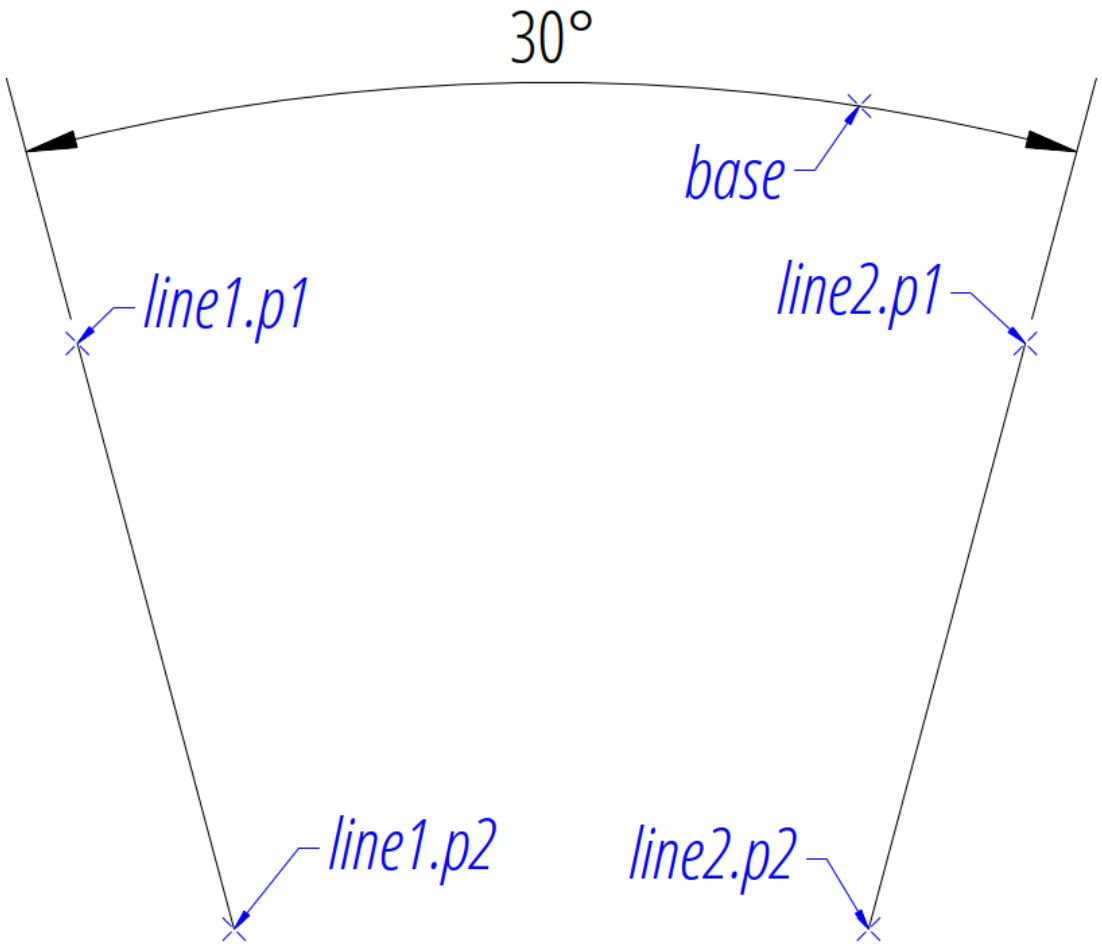
The next example shows an angular dimension for an angle defined by two lines:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

# Setup the geometric parameters for the DIMENSION entity:
base = (5.8833, -6.3408)  # location of the dimension line
p1 = (2.0101, -7.5156)  # start point of 1st leg
p2 = (2.7865, -10.4133)  # end point of 1st leg
p3 = (6.7054, -7.5156)  # start point of 2nd leg
p4 = (5.9289, -10.4133)  # end point of 2nd leg

# Draw the lines for visualization, not required to create the
# DIMENSION entity:
msp.add_line(p1, p2)
msp.add_line(p3, p4)

# Add an angular DIMENSION defined by two lines, the measurement text is
# placed at the default location above the dimension line:
dim = msp.add_angular_dim_2l(
    base=base,  # defines the location of the dimension line
    line1=(p1, p2),  # start leg of the angle
    line2=(p3, p4),  # end leg of the angle
    dimstyle="EZ_CURVED",  # default angular dimension style
)

# Necessary second step to create the dimension line geometry:
dim.render()
doc.saveas("angular_dimension_2l.dxf")
```

The example above creates an angular `Dimension` entity to measures the angle between two lines (*line1* and *line2*).

The *base* point defines the location of the dimension line (arc), any point on the dimension line is valid. The points *p1* and *p2* define the first leg of the angle, *p1* also defines the start point of the first extension line. The points *p3* and *p4* define the second leg of the angle and point *p3* also defines the start point of the second extension line.

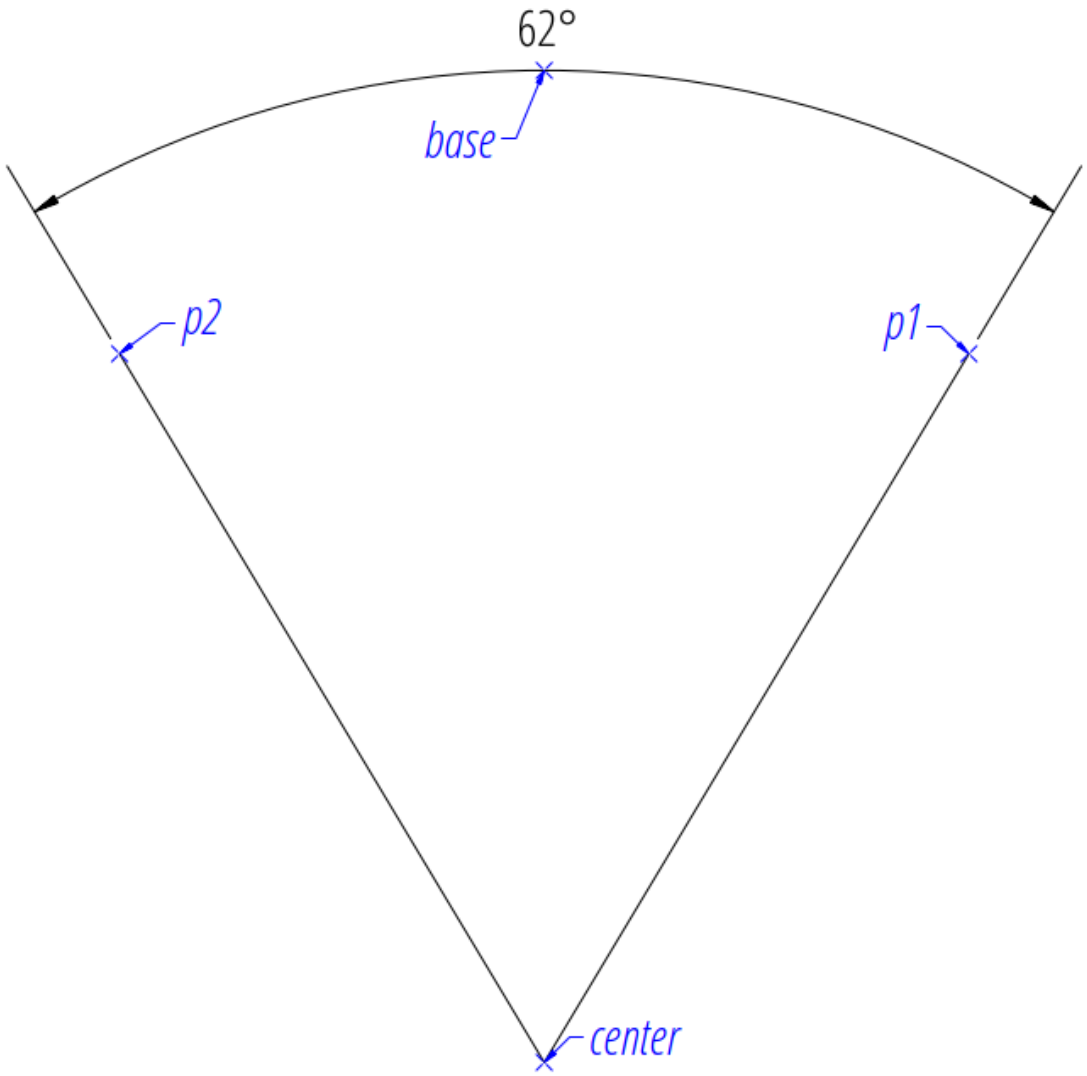
The measurement of the DIMENSION entity is the angle enclosed by the first and the second leg and where the dimension line passes the *base* point.



Angle by 3 Points

The next example shows an angular dimension defined by three points, a center point and the two end points of the angle legs:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

msp.add_angular_dim_3p(
    base=(0, 7),  # location of the dimension line
    center=(0, 0),  # center point
    p1=(-3, 5),  # end point of 1st leg = start angle
    p2=(3, 5),  # end point of 2nd leg = end angle
).render()
```



Angle from ConstructionArc

The *ezdxf.math.ConstructionArc* provides various class methods for creating arcs and the construction tool can be created from an ARC entity.

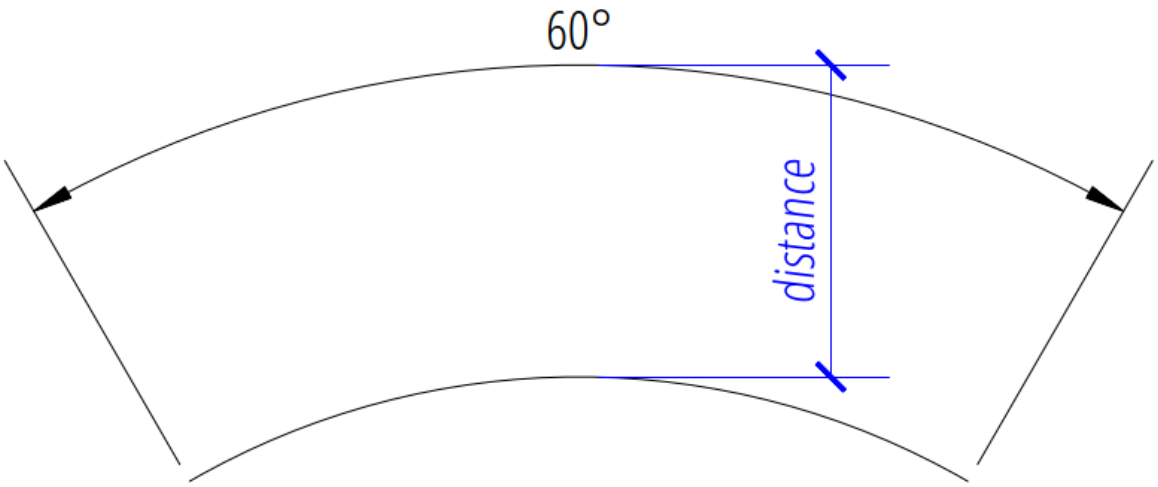
Add an angular dimension to an ARC entity:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

arc = msp.add_arc(
    center=(0, 0),
    radius=5,
    start_angle = 60,
```

(continues on next page)

(continued from previous page)

```
    end_angle = 120,
)
msp.add_angular_dim_arc(
    arc.construction_tool(),
    distance=2,
).render()
```



Placing Measurement Text

The default location of the measurement text depends on various `DimStyle` parameters and is applied if no user defined text location is defined.

Note: Not all possibles features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the angular dimension there are less features implemented than for the linear dimension because of the lack of good documentation.

See also:

- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file standards.py shows how to create your own DIMSTYLES.
- The Script dimension_angular.py shows examples for angular dimensions.

Default Text Locations

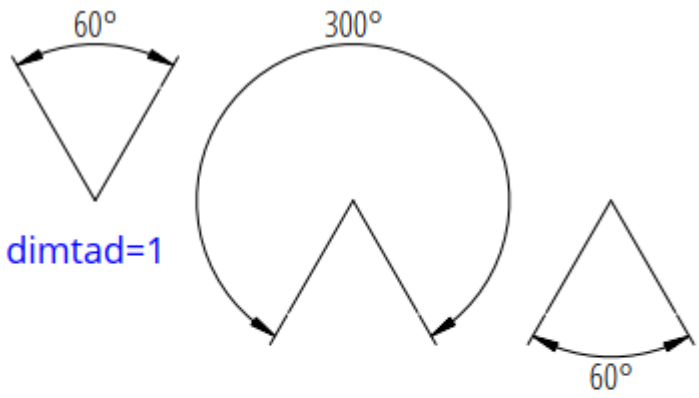
The DIMSTYLE "EZ_CURVED" places the measurement text in the center of the angle above the dimension line. The first examples above show the measurement text at the default text location.

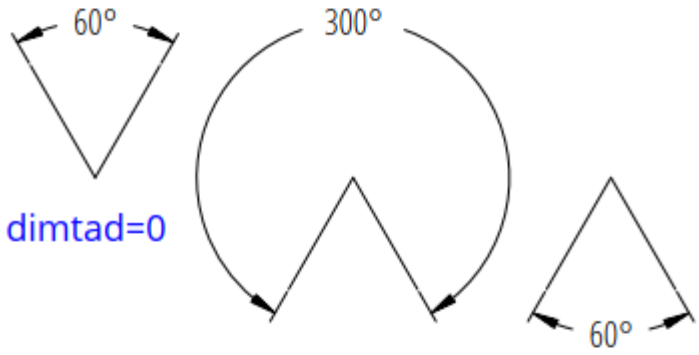
The text direction angle is always perpendicular to the line from the text center to the center point of the angle unless this angle is manually overridden.

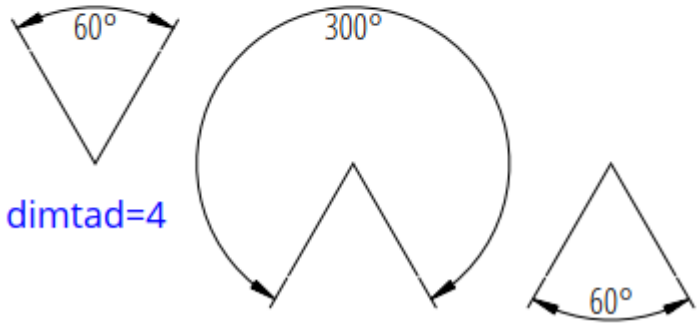
The **"vertical"** location of the measurement text relative to the dimension line is defined by `dimtad`:

0	Center, it is possible to adjust the vertical location by `dimtvp`
1	Above
2	Outside, handled like *Above* by *ezdxf*
3	JIS, handled like *Above* by *ezdxf*
4	Below

```
msp.add_angular_dim_cra(
    center=(3, 3),
    radius=3,
    distance=1,
    start_angle=60,
    end_angle=120,
    override={
        "dimtad": 1,  # 0=center; 1=above; 4=below;
    },
).render()
```





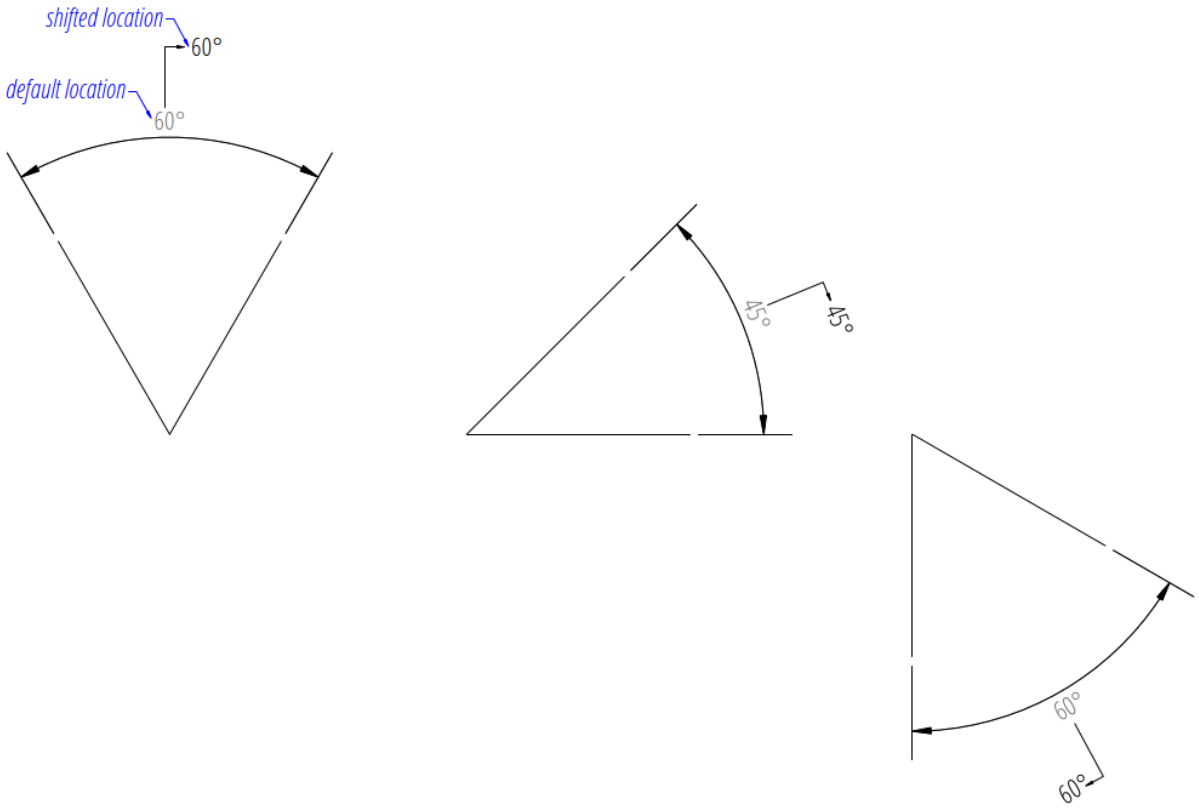


Arrows and measurement text are placed "outside" automatically if the available space between the extension lines isn't sufficient. This overrides the `dimtad` value by 1 ("above"). *Ezdxf* follows its own rules, ignores the `dimatfit` attribute and works similar to `dimatfit` = 1, move arrows first, then text:

Shift Text From Default Location

The method `shift_text()` shifts the measurement text away from the default location. The shifting direction is aligned to the text rotation of the default measurement text.

```
dim = msp.add_angular_dim_cra(
    center=(3, 3),
    radius=3,
    distance=1,
    start_angle=60,
    end_angle=120,
)
# shift text from default text location:
dim.shift_text(0.5, 1.0)
dim.render()
```



This is just a rendering effect, editing the dimension line in a CAD application resets the text to the default location.

User Defined Text Locations

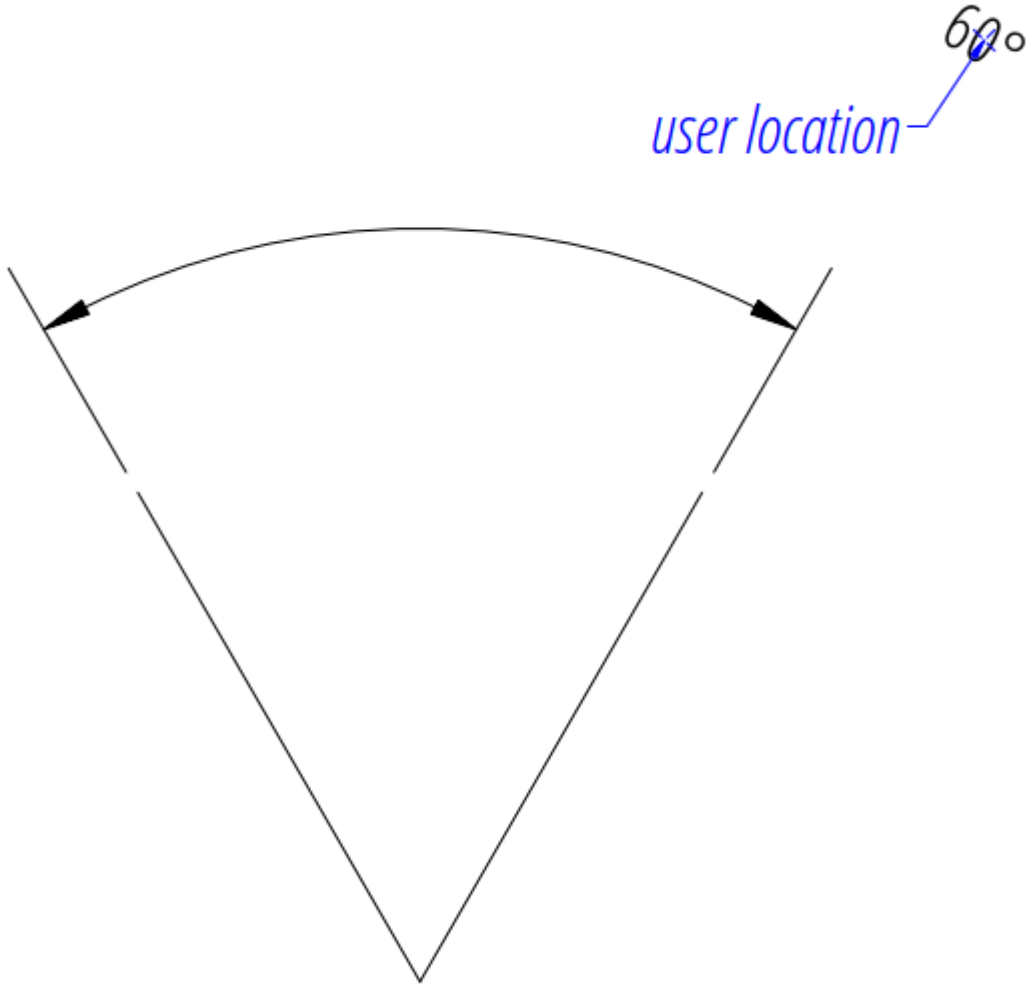
Beside the default location it is always possible to override the text location by a user defined location.

The coordinates of user locations are located in the rendering UCS and the default rendering UCS is the *WCS*.

Absolute User Location

Absolute placing of the measurement text means relative to the origin of the render UCS. The user location is stored in the DIMENSION entity, which means editing the dimension line in a CAD application does not alter the text location. This location also determines the rotation of the measurement text.

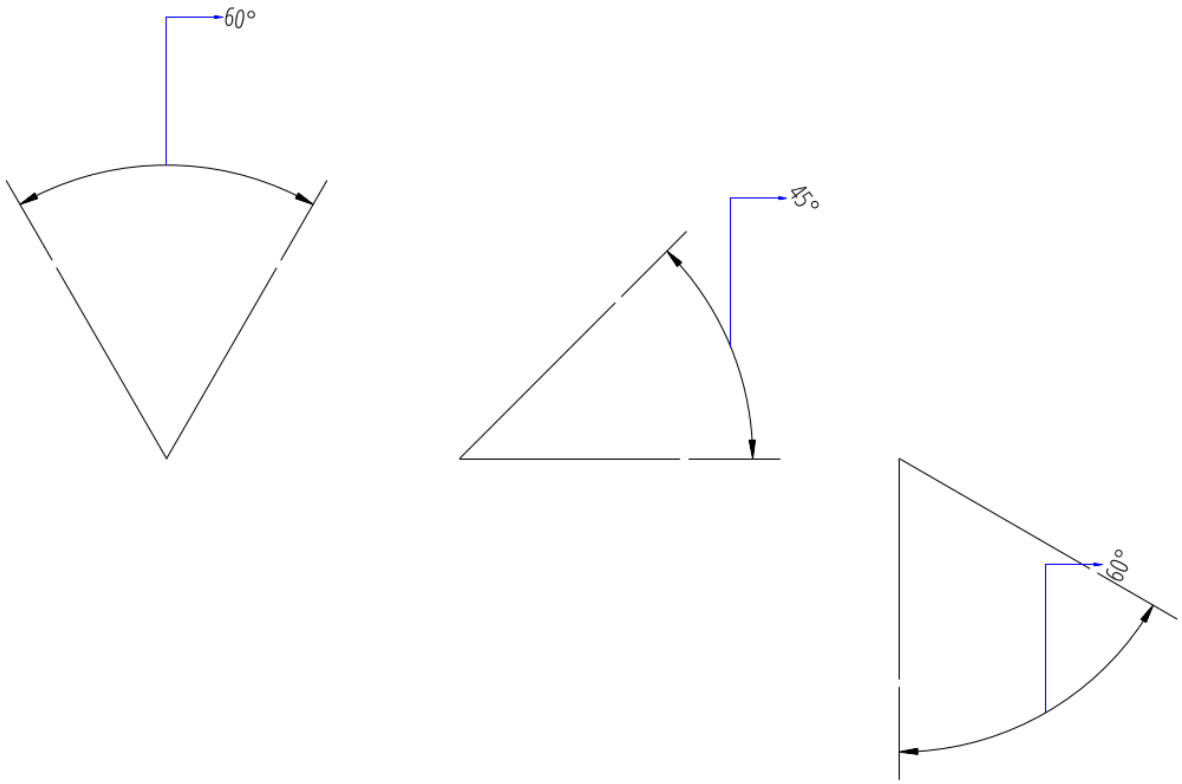
```
dim = msp.add_angular_dim_cra(
    center=(3, 3),
    radius=3,
    distance=1,
    start_angle=60,
    end_angle=120,
    location=(5, 8),  # user defined measurement text location
)
dim.render()
```



Relative User Location

Relative placing of the measurement text means relative to the middle of the dimension line. This is only possible by calling the `set_location()` method, and the argument *relative* has to be `True`. The user location is stored in the DIMENSION entity, which means editing the dimension line in a CAD application does not alter the text location. This location also determines the rotation of the measurement text.

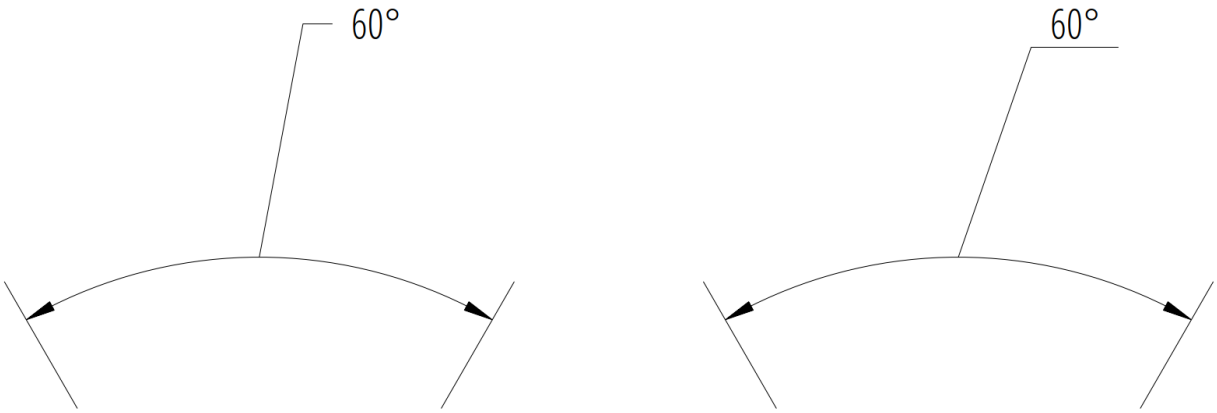
```
dim = msp.add_angular_dim_cra(
    center=(3, 3),
    radius=3,
    distance=1,
    start_angle=60,
    end_angle=120,
)
dim.set_location((1, 2), relative=True)
dim.render()
```



Adding a Leader

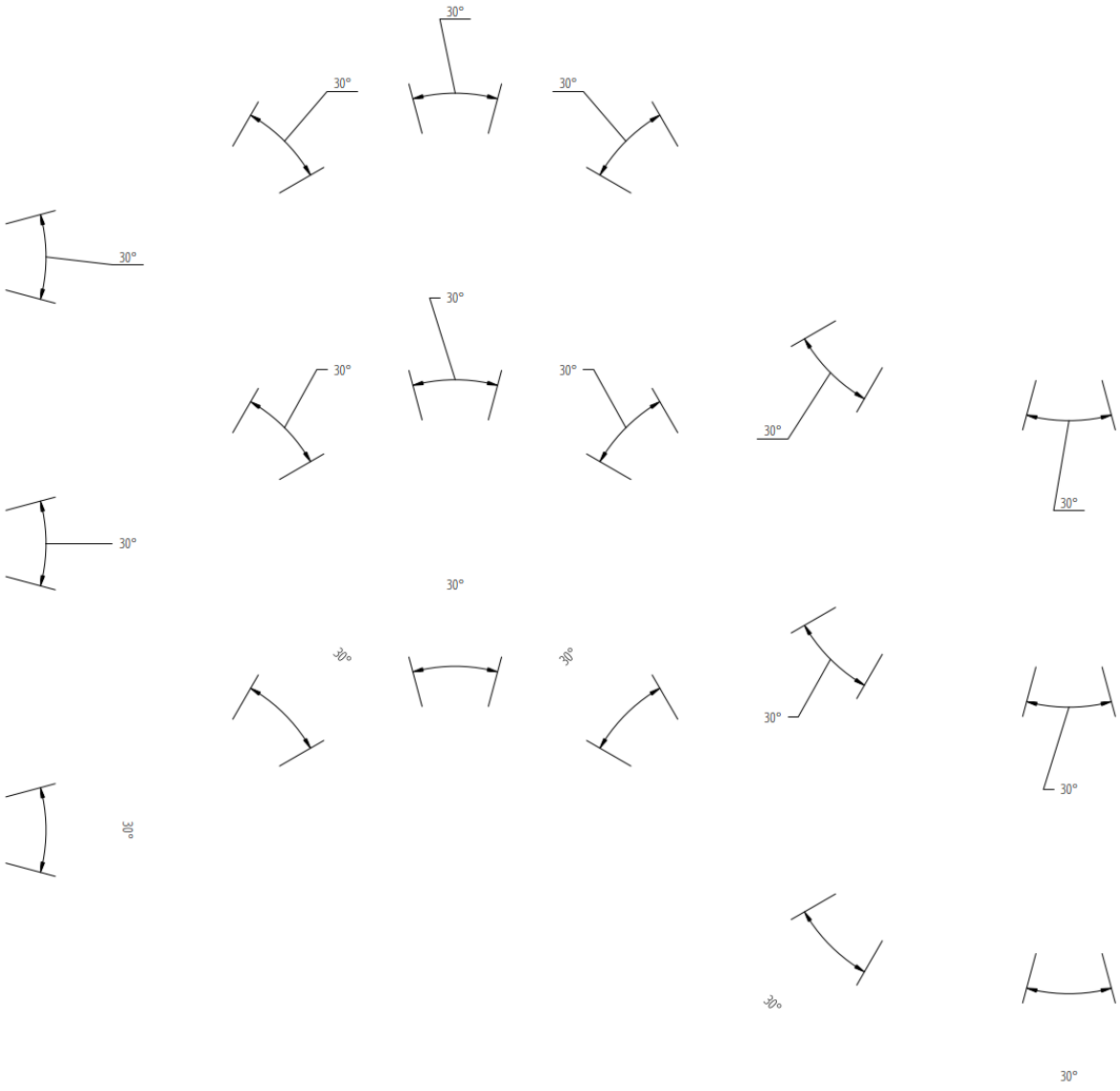
The method `set_location()` has the option to add a leader line to the measurement text. This also aligns the text rotation to the render UCS x-axis, this means in the default case the measurement text is horizontal. The leader line can be "below" the text or start at the "left" or "right" center of the text, this location is defined by the *dimtad* attribute, 0 means "center" and any value != 0 means "below".

```
for dimtad, x in [(0, 0), (4, 6)]:
    dim = msp.add_angular_dim_cra(
        center=(3 + x, 3),
        radius=3,
        distance=1,
        start_angle=60,
        end_angle=120,
        override={"dimtad": dimtad}  # "center" == 0; "below" != 0;
    )
    dim.set_location((1, 2), relative=True, leader=True)
    dim.render()
```



Advanced version which calculates the relative text location: The user location vector has a length 2 and the orientation is defined by *center_angle* pointing away from the center of the angle.

```
import ezdxf
from ezdxf.math import Vec3

doc = ezdxf.new(setup=True)
msp = doc.modelspace()
for dimtad, y, leader in [
    [0, 0, False],
    [0, 7, True],
    [4, 14, True],
]:
    for x, center_angle in [
        (0, 0), (7, 45), (14, 90), (21, 135), (26, 225), (29, 270)
    ]:
        dim = msp.add_angular_dim_cra(
            center=(x, y),
            radius=3.0,
            distance=1.0,
            start_angle=center_angle - 15.0,
            end_angle=center_angle + 15.0,
            override={"dimtad": dimtad},
        )
        # The user location is relative to the center of the dimension line:
        usr_location = Vec3.from_deg_angle(angle=center_angle, length=2.0)
        dim.set_location(usr_location, leader=leader, relative=True)
        dim.render()
```



Overriding Text Rotation

All factory methods supporting the argument *text_rotation* can override the measurement text rotation. The user defined rotation is relative to the render UCS x-axis (default is WCS).

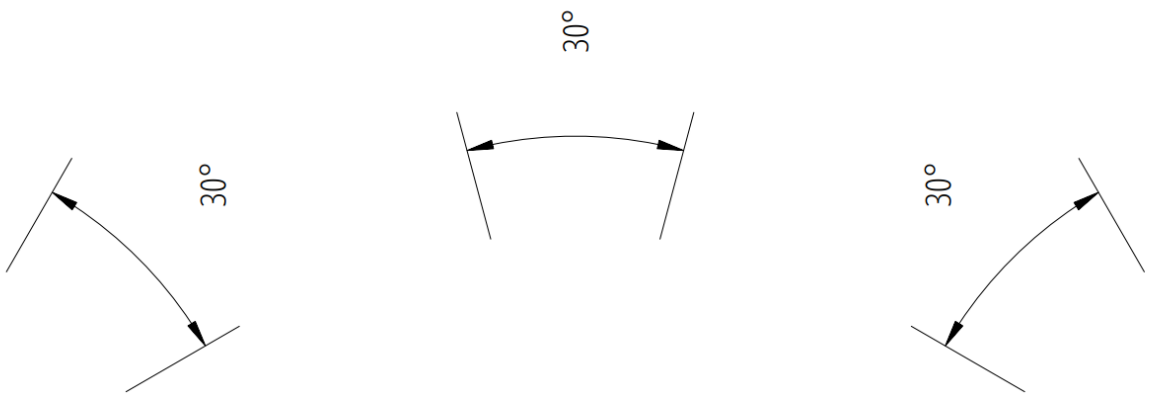
This example uses a relative text location without a leader and forces the text rotation to 90 degrees:

```
for x, center_angle in [(7, 45), (14, 90), (21, 135)]:
    dim = msp.add_angular_dim_cra(
        center=(x, 0),
        radius=3.0,
        distance=1.0,
        start_angle=center_angle - 15.0,
        end_angle=center_angle + 15.0,
        text_rotation=90,  # vertical text
    )
```

(continues on next page)

(continued from previous page)

```
    usr_location = Vec3.from_deg_angle(angle=center_angle, length=1.0)
    dim.set_location(usr_location, leader=False, relative=True)
    dim.render()
```



Angular Units

Angular units are set by *dimaunit*:

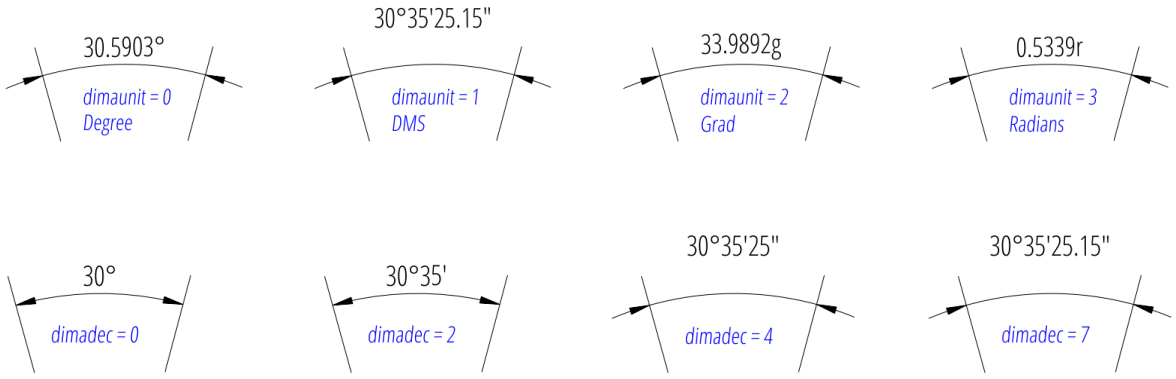
0	Decimal degrees
1	Degrees/Minutes/Seconds, dimadec controls the shown precision • dimadec=0: 30° • dimadec=2: 30°35' • dimadec=4: 30°35'25" • dimadec=7: 30°35'25.15"
2	Grad
3	Radians

```
d1 = 15
d2 = 15.59031944
for x, (dimaunit, dimadec) in enumerate(
    [
        (0, 4),
        (1, 7),
        (2, 4),
        (3, 4),
    ]
):
    dim = msp.add_angular_dim_cra(
        center=(x * 4.0, 0.0),
        radius=3.0,
        distance=1.0,
        start_angle=90.0 - d1,
```

(continues on next page)

(continued from previous page)

```
        end_angle=90.0 + d2,
        override={
            "dimaunit": dimaunit,
            "dimadec": dimadec,
        },
    )
    dim.render()
```



Overriding Measurement Text

See Linear Dimension Tutorial: *Overriding Measurement Text*

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: *Measurement Text Formatting and Styling*

Tolerances and Limits

See Linear Dimension Tutorial: *Tolerances and Limits*

6.5.27 Tutorial for Arc Dimensions

Please read the *Tutorial for Linear Dimensions* before, if you haven't. This is a repetition of the *Tutorial for Angular Dimensions*, because *ezdxf* reuses the angular dimension to render arc dimensions. This approach is very different to CAD applications, but also much less work.

Note: *Ezdxf* does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Dimension Style "EZ_CURVED"

All factory methods to create arc dimensions uses the dimension style "EZ_CURVED" for curved dimension lines which is defined as:

- angle unit is decimal degrees, `dimaunit` = 0
- measurement text height = 0.25 (drawing scale = 1:100)
- measurement text location is above the dimension line
- closed filled arrow and arrow size `dimasz` = 0.25
- `dimzin` = 2, suppresses trailing zeros (e.g. 12.5000 becomes 12.5)
- `dimarcsym` = 2, disables the arc symbol, 0 renders only an open round bracket "(" in front of the text and 1 for arc symbol above the text is not supported, renders like disabled

For more information go to: *Dimension Style "EZ_CURVED"*

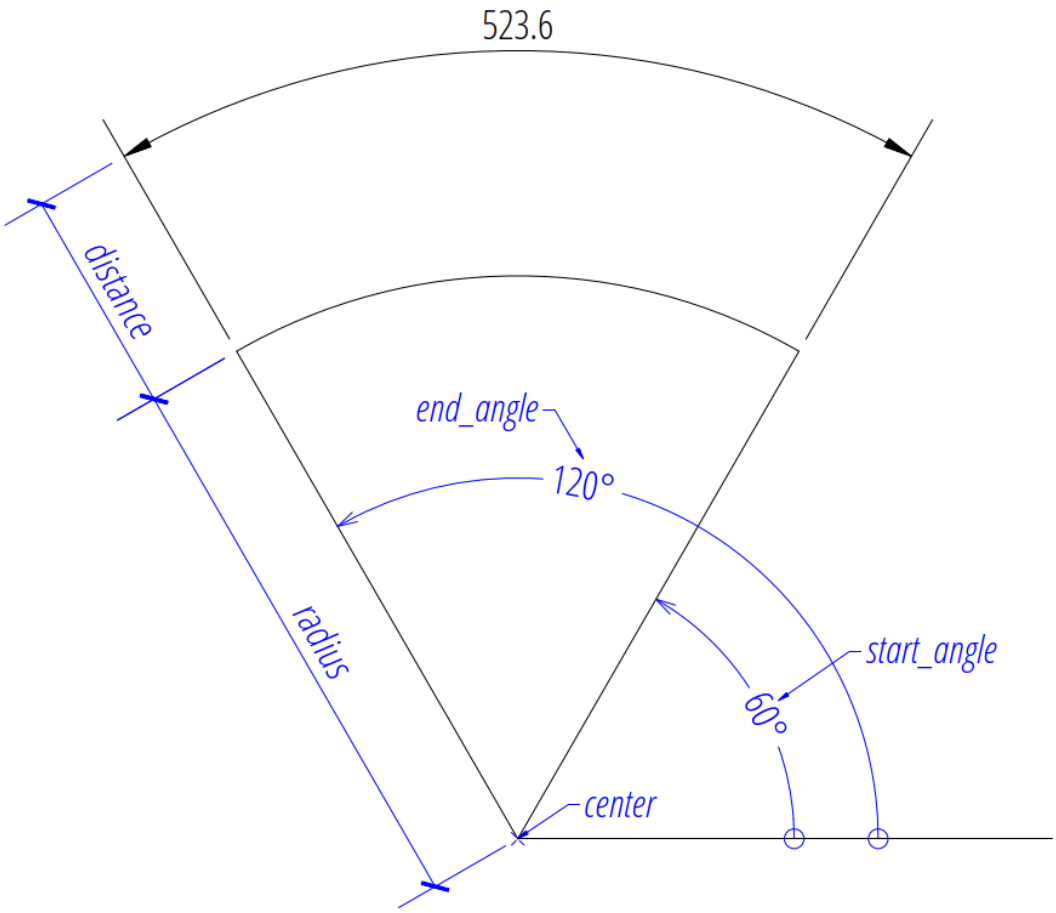
Factory Methods to Create Arc Dimensions

Defined by Center, Radius and Angles

The first example shows an arc dimension defined by the center point, radius, start- and end angles:

```
import ezdxf

# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new(setup=True)

# Add new entities to the modelspace:
msp = doc.modelspace()

# Add an arc DIMENSION defined by the center point, start- and end angles,
# the measurement text is placed at the default location above the dimension
# line:
dim = msp.add_arc_dim_cra(
    center=(5, 5),  # center point of the angle
    radius=5,  # distance from center point to the start of the extension lines
    start_angle=60,  # start angle in degrees
    end_angle=120,  # end angle in degrees
    distance=2,  # distance from start of the extension lines to the dimension line
    dimstyle="EZ_CURVED",  # default angular dimension style
)

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
dim.render()
doc.saveas("arc_dimension_cra.dxf")
```

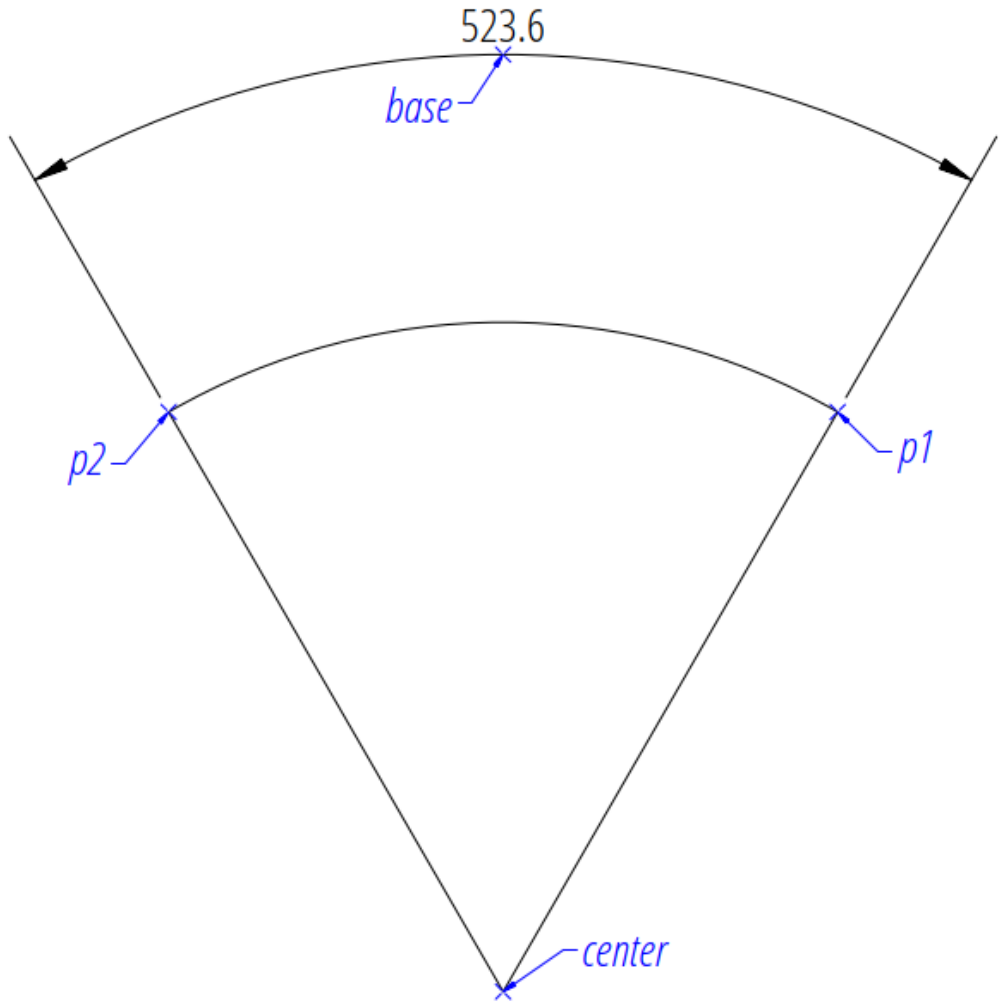
The return value *dim* is **not** a dimension entity, instead a *DimStyleOverride* object is returned, the dimension entity is stored as `dim.dimension`.



Arc by 3 Points

The next example shows an angular dimension defined by three points, a center point and the two end points of the angle legs, the first point defines the radius, the second point defines only the end angle, the distance from the center point is not relevant:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

msp.add_arc_dim_3p(
    base=(0, 7),  # location of the dimension line
    center=(0, 0),  # center point
    p1=(2.5, 4.330127018922193),  # 1st point of arc defines start angle and radius
    p2=(-2.5, 4.330127018922194),  # 2nd point defines the end angle
).render()
```



Angle from ConstructionArc

The *ezdxf.math.ConstructionArc* provides various class methods for creating arcs and the construction tool can be created from an ARC entity.

Add an angular dimension to an ARC entity:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

arc = msp.add_arc(
    center=(0, 0),
    radius=5,
    start_angle = 60,
    end_angle = 120,
```

(continues on next page)

(continued from previous page)

```
)
msp.add_arc_dim_arc(
    arc.construction_tool(),
    distance=2,
).render()
```

Placing Measurement Text

The default location of the measurement text depends on various `DimStyle` parameters and is applied if no user defined text location is defined.

Note: Not all possibles features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the arc dimension there are less features implemented than for the linear dimension because of the lack of good documentation. If the arc symbol is enabled (`dimarcsym` = 0) only an open round bracket "(" is rendered in front of the measurement text!

See also:

- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file standards.py shows how to create your own DIMSTYLES.
- The Script dimension_arc.py shows examples for angular dimensions.

Default Text Locations

The DIMSTYLE "EZ_CURVED" places the measurement text in the center of the angle above the dimension line. The first examples above show the measurement text at the default text location.

The text direction angle is always perpendicular to the line from the text center to the center point of the angle unless this angle is manually overridden.

Arrows and measurement text are placed "outside" automatically if the available space between the extension lines isn't sufficient.

For more information go to: *Default Text Locations*

Shift Text From Default Location

The method `shift_text()` shifts the measurement text away from the default location. The shifting direction is aligned to the text rotation of the default measurement text.

For more information go to: *Shift Text From Default Location*

User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location.

The coordinates of user locations are located in the rendering UCS and the default rendering UCS is the *WCS*.

For more information go to: *User Defined Text Locations*

Absolute User Location

Absolute placing of the measurement text means relative to the origin of the render UCS.

For more information go to: *User Defined Text Locations*

Relative User Location

Relative placing of the measurement text means relative to the middle of the dimension line.

For more information go to: *User Defined Text Locations*

Adding a Leader

Add a leader line to the measurement text and set the text rotation to "horizontal".

For more information go to: *User Defined Text Locations*

Overriding Text Rotation

All factory methods supporting the argument *text_rotation* can override the measurement text rotation. The user defined rotation is relative to the render UCS x-axis (default is WCS).

For more information go to: *User Defined Text Locations*

Overriding Measurement Text

See Linear Dimension Tutorial: *Overriding Text Rotation*

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: *Measurement Text Formatting and Styling*

Tolerances and Limits

See Linear Dimension Tutorial: *Tolerances and Limits*

6.5.28 Tutorial for Ordinate Dimensions

Please read the *Tutorial for Linear Dimensions* before, if you haven't.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Local Coordinate System

Ordinate dimensioning is used when the x- and the y-coordinates from a location (feature), are the only dimensions necessary. The dimensions to each feature, originate from one datum location, called "origin" in this tutorial.

The local coordinate system (LCS) in which the measurement is done, is defined by the *origin* and the *rotation* angle around the z-axis in the rendering UCS, which is the *WCS* by default.

Factory Methods to Create Ordinate Dimensions

All factory methods for creating ordinate dimensions expect global coordinates to define the feature location.

Global Feature Location

The first example shows ordinate dimensions defined in the render UCS, in this example the *WCS*, this is how the DIMENSION entity expects the coordinates of the feature location:

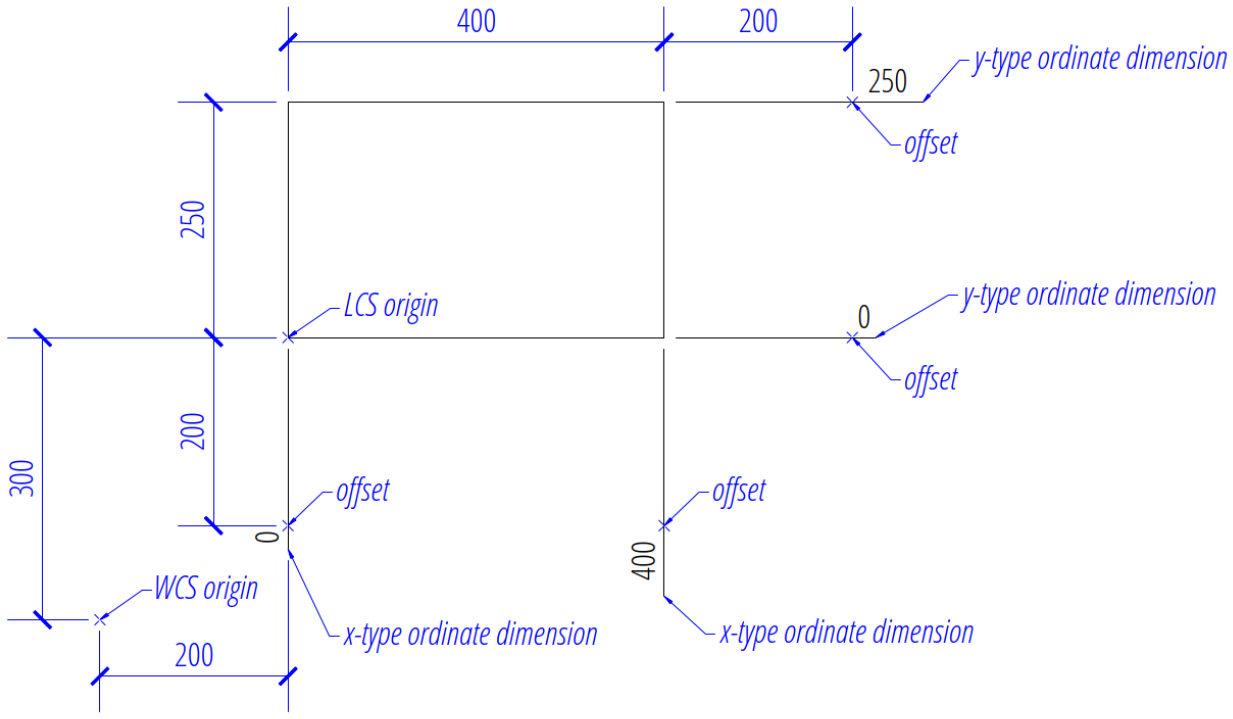
```
import ezdxf
from ezdxf.math import Vec3
from ezdxf.render import forms

# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new(setup=True)

# Add new entities to the modelspace:
msp = doc.modelspace()
# Add a rectangle: width=4, height = 2.5, lower left corner is WCS(x=2, y=3)
origin = Vec3(2, 3)
msp.add_lwpolyline(
    forms.translate(forms.box(4, 2.5), origin),
    close=True
)

# Add an x-type ordinate DIMENSION with global feature locations:
msp.add_ordinate_x_dim(
    # lower left corner
    feature_location=origin + (0, 0),  # feature location in the WCS
    offset=(0, -2),  # end of leader, relative to the feature location
    origin=origin,
).render()
msp.add_ordinate_x_dim(
```

(continues on next page)

(continued from previous page)

```
    # lower right corner
    feature_location=origin + (4, 0),  # feature location in the WCS
    offset=(0, -2),
    origin=origin,
).render()

# Add an y-type ordinate DIMENSION with global feature locations:
msp.add_ordinate_y_dim(
    # lower right corner
    feature_location=origin + (4, 0),  # feature location in the WCS
    offset=(2, 0),
    origin=origin,
).render()
msp.add_ordinate_y_dim(
    # upper right corner
    feature_location=origin + (4, 2.5),  # feature location in the WCS
    offset=(2, 0),
    origin=origin,
).render()

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
doc.saveas("ord_global_features.dxf")
```

The return value *dim* is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as `dim.dimension`.



Local Feature Location

The previous examples shows that the calculation of the global feature location is cumbersome and it gets even more complicated for a rotated LCS.

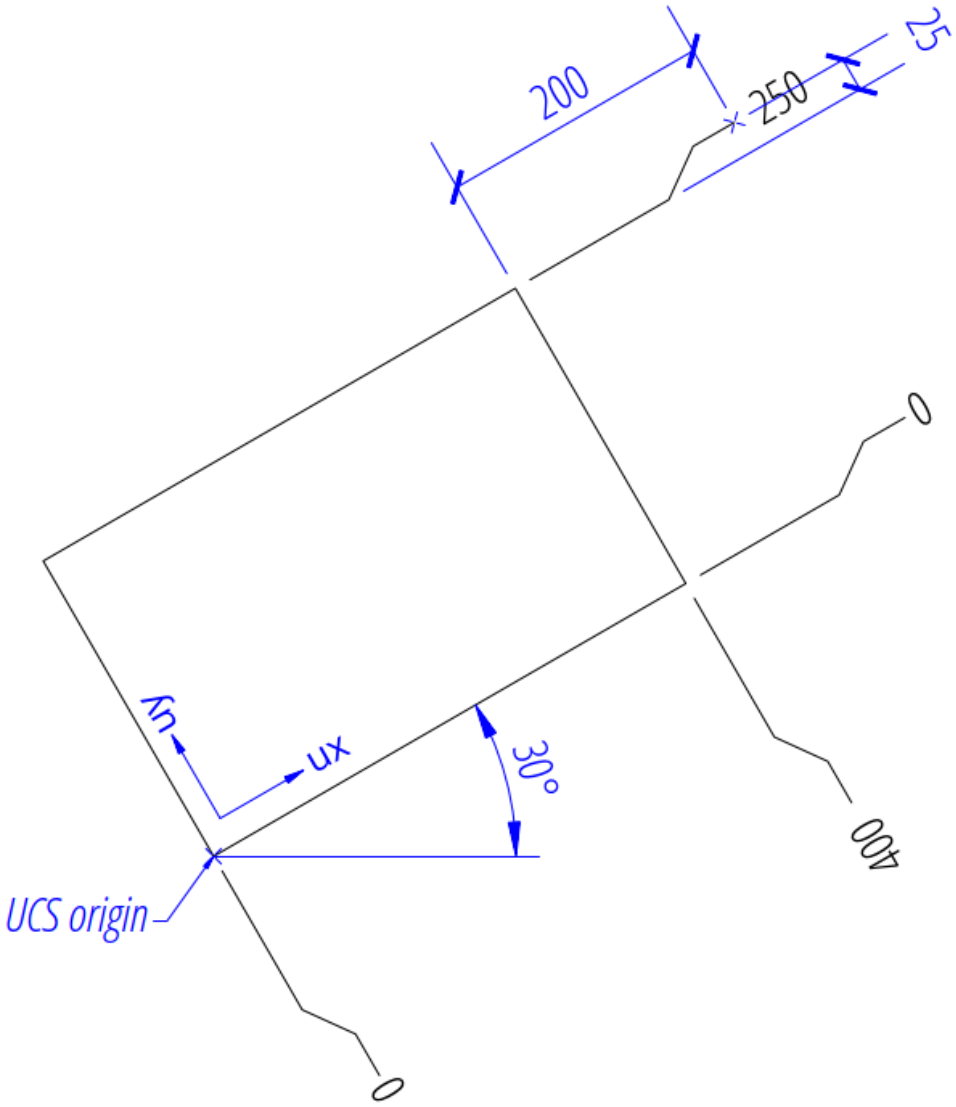
This example shows how to use a render *UCS* for using locale coordinates to define the feature locations:

```
import ezdxf
from ezdxf.math import Vec3, UCS
from ezdxf.render import forms

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

# Create a special DIMSTYLE for "vertical" centered measurement text:
dimstyle = doc.dimstyles.duplicate_entry("EZDXF", "ORD_CENTER")
dimstyle.dxf.dimtad = 0  # "vertical" centered measurement text

# Add a rectangle: width=4, height = 2.5, lower left corner is WCS(x=2, y=3),
# rotated about 30 degrees:
origin = Vec3(2, 3)
msp.add_lwpolyline(
    forms.translate(forms.rotate(forms.box(4, 2.5), 30), origin),
    close=True
)

# Define the rotated local render UCS.
# The origin is the lower-left corner of the rectangle and the axis are
# aligned to the rectangle edges:
# The y-axis "uy" is calculated automatically by the right-hand rule.
ucs = UCS(origin, ux=Vec3.from_deg_angle(30), uz=(0, 0, 1))

# Add a x-type ordinate DIMENSION with local feature locations:
# the origin is now the origin of the UCS, which is (0, 0) the default value of
# "origin" and the feature coordinates are located in the UCS:
msp.add_ordinate_x_dim(
    # lower left corner
    feature_location=(0, 0),  # feature location in the UCS
    offset=(0.25, -2),  # # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs)  # Important when using a render UCS!
msp.add_ordinate_x_dim(
    # lower right corner
    feature_location=(4, 0),  # feature location in the UCS
    offset=(0.25, -2),  # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs)  # Important when using a render UCS!

# Add a y-type ordinate DIMENSION with local feature coordinates:
msp.add_ordinate_y_dim(
    # lower right corner
    feature_location=(4, 0),  # feature location in the UCS
    offset=(2, 0.25),  # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs)  # Important when using a render UCS!
msp.add_ordinate_y_dim(
    # upper right corner
    feature_location=(4, 2.5),  # feature location in the UCS
```

(continues on next page)

(continued from previous page)

```
    offset=(2, 0.25),  # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs)  # Important when using a render UCS!
doc.saveas("ord_local_features.dxf")
```



Placing Measurement Text

The *ezdxf* ordinate DIMENSION renderer places the measurement text always at the default location, because the location of the leader end point is given by the argument *offset* in the factory methods, which provides a flexible way to place the measurement text, overriding the text location by an explicit user location is not supported, also the user text rotation is not supported, the text is always aligned to the local coordinate system x- and y-axis.

See also:

- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file standards.py shows how to create your own DIMSTYLES.

- The Script dimension_ordinate.py shows examples for angular dimensions.

Overriding Measurement Text

See Linear Dimension Tutorial: *Overriding Text Rotation*

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: *Measurement Text Formatting and Styling*

Tolerances and Limits

See Linear Dimension Tutorial: *Tolerances and Limits*

6.5.29 Tutorial for the Geo Add-on

This tutorial shows how to load a GPS track into a geo located DXF file and also the inverse operation, exporting geo located DXF entities as GeoJSON files.

Please read the section *Intended Usage* in the documentation of the `ezdxf.addons.geo` module first.

> **Warning:** TO ALL BEGINNERS!
>
> If you are just learning to work with geospatial data, using DXF files is not the way to go! DXF is not the first choice for storing data for spatial data analysts. If you run into problems I cannot help you as I am just learning myself.

The complete source code and test data for this tutorial are available in the github repository:

https://github.com/mozman/ezdxf/tree/master/docs/source/tutorials/src/geo

Setup Geo Location Reference

The first step is setting up the geo location reference, which is **not** doable with ezdxf yet - this feature may come in the future - but for now you have to use a CAD application to do this. If the DXF file has no geo location reference the projected 2D coordinates are most likely far away from the WCS origin (0, 0), use the CAD command "ZOOM EXTENDS" to find the data.

Load GPX Data

The GPX format stores GPS data in a XML format, use the `ElementTree` class to load the data:

```
def load_gpx_track(p: Path) -> Iterable[Tuple[float, float]]:
    """Load all track points from all track segments at once."""
    gpx = ET.parse(p)
    root = gpx.getroot()
    for track_point in root.findall(".//gpx:trkpt", GPX_NS):
        data = track_point.attrib
        # Elevation is not supported by the geo add-on.
        yield float(data["lon"]), float(data["lat"])
```

The loaded GPS data has a WSG84 EPSG:4326 projection as longitude and latitude in decimal degrees. The next step is to create a `GeoProxy` object from this data, the `GeoProxy.parse()` method accepts a `__geo_interface__` mapping or a Python object with a `__geo_interface__` attribute/property. In this case as simple "LineString" object for all GPS points is sufficient:

```
def add_gpx_track(msp, track_data, layer: str):
    geo_mapping = {
        "type": "LineString",
        "coordinates": track_data,
    }
    geo_track = geo.GeoProxy.parse(geo_mapping)
```

Transform the data from the polar representation EPSG:4326 into a 2D cartesian map representation EPSG:3395 called "World Mercator", this is the only projection supported by the add-on, without the need to write a custom transformation function:

```
    geo_track.globe_to_map()
```

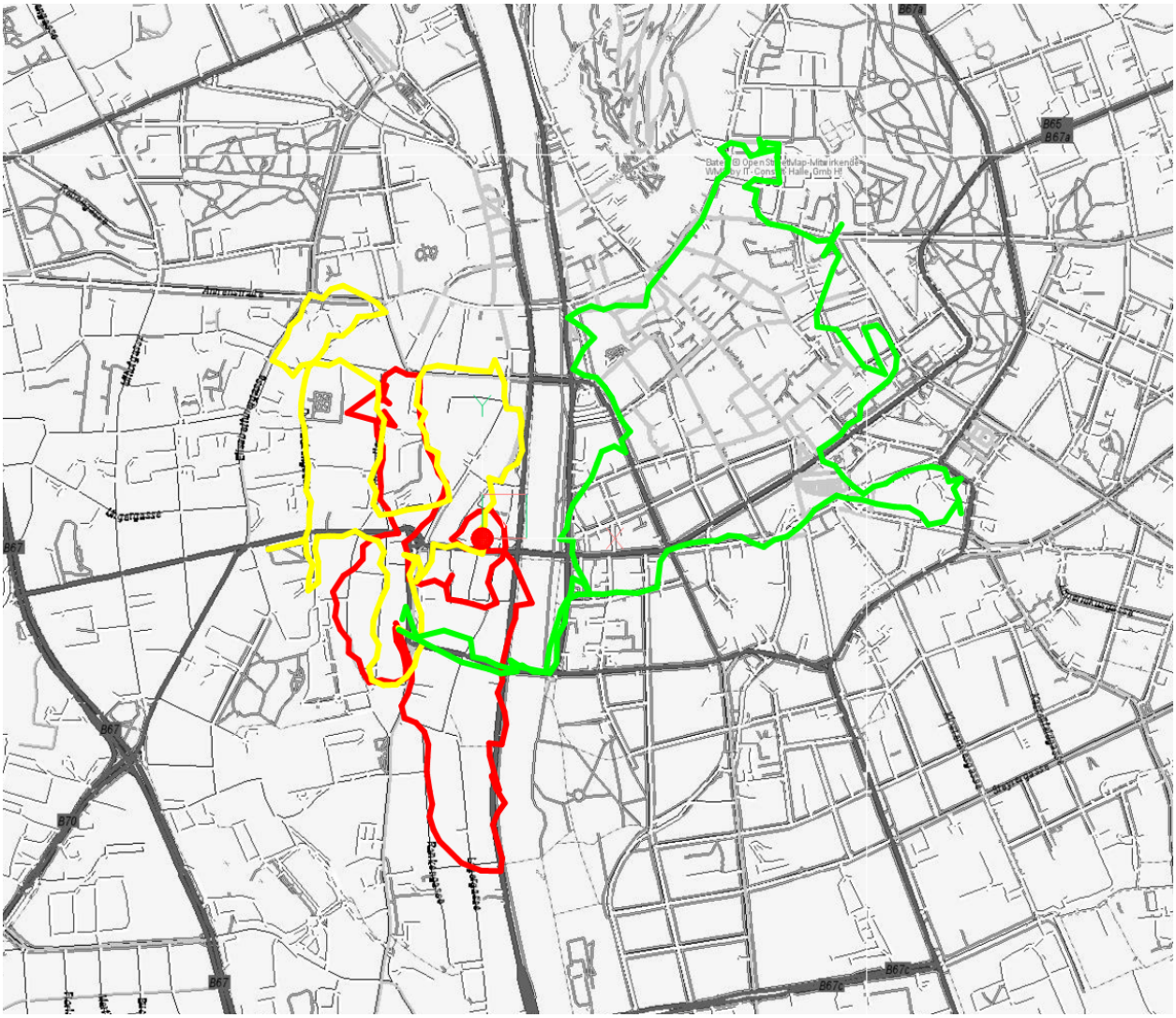
The data is now transformed into 2D cartesian coordinates in meters and most likely far away from origin (0, 0), the data stored in the GEODATA entity helps to transform the data into the DXF WCS in modelspace units, if the DXF file has no geo location reference you have to stick with the large coordinates:

```
    # Load geo data information from the DXF file:
    geo_data = msp.get_geodata()
    if geo_data:
        # Get the transformation matrix and epsg code:
        m, epsg = geo_data.get_crs_transformation()
    else:
        # Identity matrix for DXF files without a geo location reference:
        m = Matrix44()
        epsg = 3395
    # Check for compatible projection:
    if epsg == 3395:
        # Transform CRS coordinates into DXF WCS:
        geo_track.crs_to_wcs(m)
        # Create DXF entities (LWPOLYLINE)
        for entity in geo_track.to_dxf_entities(dxfattribs={"layer": layer}):
            # Add entity to the modelspace:
            msp.add_entity(entity)
    else:
        print(f"Incompatible CRS EPSG:{epsg}")
```

We are ready to save the final DXF file:

```
doc.saveas(str(out_path))
```

In BricsCAD the result looks like this, the underlying images were added by the BricsCAD command MAPCONNECT and such a feature is **not** planned for the add-on:



Export DXF Entities as GeoJSON

This will only work with a proper geo location reference, the code shown accepts also WCS data from DXF files without a GEODATA object, but the result is just unusable - but in valid GeoJSON notation.

First get epsg code and the CRS transformation matrix:

```
    # Get the geo location information from the DXF file:
    geo_data = msp.get_geodata()
    if geo_data:
        # Get transformation matrix and epsg code:
        m, epsg = geo_data.get_crs_transformation()
    else:
        # Identity matrix for DXF files without geo reference data:
        m = Matrix44()
```

Query the DXF entities to export:

```
    for track in msp.query("LWPOLYLINE"):
        export_geojson(track, m)
```

Create a `GeoProxy` object from the DXF entity:

```
def export_geojson(entity, m):
    # Convert DXF entity into a GeoProxy object:
    geo_proxy = geo.proxy(entity)
```

Transform DXF WCS coordinates in modelspace units into the CRS coordinate system by the transformation matrix *m*:

```
    # Transform DXF WCS coordinates into CRS coordinates:
    geo_proxy.wcs_to_crs(m)
```

The next step assumes a EPSG:3395 projection, everything else needs a custom transformation function:

```
    # Transform 2D map projection EPSG:3395 into globe (polar)
    # representation EPSG:4326
    geo_proxy.map_to_globe()
```

Use the `json` module from the Python standard library to write the GeoJSON data, provided by the `GeoProxy.__geo_interface__` property:

```
    # Export GeoJSON data:
    name = entity.dxf.layer + ".geojson"
    with open(TRACK_DATA / name, "wt", encoding="utf8") as fp:
        json.dump(geo_proxy.__geo_interface__, fp, indent=2)
```

The content of the GeoJSON file looks like this:

```
{
  "type": "LineString",
  "coordinates": [
    [
      15.430999,
      47.06503
    ],
    [
      15.431039,
      47.064797
    ],
    [
      15.431206,
      47.064582
    ],
    [
      15.431283,
      47.064342
    ],
    ...
}
```

Custom Transformation Function

This sections shows how to use the GDAL package to write a custom transformation function. The example reimplements the builtin transformation from unprojected WGS84 coordinates to 2D map coordinates EPSG:3395 "World Mercator":

```
from osgeo import osr
from ezdxf.math import Vec3

# GPS track in WGS84, load_gpx_track() code see above
gpx_points = list(load_gpx_track('track1.gpx'))

# Create source coordinate system:
src_datum = osr.SpatialReference()
src_datum.SetWellKnownGeoCS('WGS84')

# Create target coordinate system:
target_datum = osr.SpatialReference()
target_datum.SetWellKnownGeoCS('EPSG:3395')

# Create transformation object:
ct = osr.CoordinateTransform(src_datum, target_datum)

# Create GeoProxy() object:
geo_proxy = GeoProxy.parse({
    'type': 'LineString',
    'coordinates': gpx_points
})

# Apply a custom transformation function to all coordinates:
geo_proxy.apply(lambda v: Vec3(ct.TransformPoint(v.x, v.y)))
```

The same example with the pyproj package:

```
from pyproj import Transformer
from ezdxf.math import Vec3

# GPS track in WGS84, load_gpx_track() code see above
gpx_points = list(load_gpx_track('track1.gpx'))

# Create transformation object:
ct = Transformer.from_crs('EPSG:4326', 'EPSG:3395')

# Create GeoProxy() object:
geo_proxy = GeoProxy.parse({
    'type': 'LineString',
    'coordinates': gpx_points
})

# Apply a custom transformation function to all coordinates:
geo_proxy.apply(lambda v: Vec3(ct.transform(v.x, v.y)))
```

Polygon Validation by Shapely

Ezdxf tries to avoid to create invalid polygons from HATCH entities like a hole in another hole, but not all problems are detected by ezdxf, especially overlapping polygons. For a reliable and robust result use the Shapely package to check for valid polygons:

```
import ezdxf
from ezdxf.addons import geo
from shapely.geometry import shape

# Load DXF document including HATCH entities.
doc = ezdxf.readfile('hatch.dxf')
msp = doc.modelspace()

# Test a single entity
# Get the first DXF hatch entity:
hatch_entity = msp.query('HATCH').first

# Create GeoProxy() object:
hatch_proxy = geo.proxy(hatch_entity)

# Shapely supports the __geo_interface__
shapely_polygon = shape(hatch_proxy)

if shapely_polygon.is_valid:
    ...
else:
    print(f'Invalid Polygon from {str(hatch_entity)}.')

# Remove invalid entities by a filter function
def validate(geo_proxy: geo.GeoProxy) -> bool:
    # Multi-entities are divided into single entities:
    # e.g. MultiPolygon is verified as multiple single Polygon entities.
    if geo_proxy.geotype == 'Polygon':
        return shape(geo_proxy).is_valid
    return True

# The gfilter() function let only pass compatible DXF entities
msp_proxy = geo.GeoProxy.from_dxf_entities(geo.gfilter(msp))

# remove all mappings for which validate() returns False
msp_proxy.filter(validate)
```

Interface to GDAL/OGR

The GDAL/OGR package has no direct support for the `__geo_interface__`, but has builtin support for the GeoJSON format:

```
from osgeo import ogr
from ezdxf.addons import geo
from ezdxf.render import random_2d_path
import json

p = geo.GeoProxy({'type': 'LineString', 'coordinates': list(random_2d_path(20))})
# Create a GeoJSON string from the __geo_interface__ object by the json
# module and feed the result into ogr:
```

(continues on next page)

(continued from previous page)

```
line_string = ogr.CreateGeometryFromJson(json.dumps(p.__geo_interface__))

# Parse the GeoJSON string from ogr by the json module and feed the result
# into a GeoProxy() object:
p2 = geo.GeoProxy.parse(json.loads(line_string.ExportToJson()))
```

6.5.30 Storing Custom Data in DXF Files

This tutorial describes how to store custom data in DXF files using standard DXF features.

Saving data in comments is not covered in this section, because comments are not a reliable way to store information in DXF files and *ezdxf* does not support adding comments to DXF files. Comments are also ignored by *ezdxf* and many other DXF libraries when loading DXF files, but there is a `ezdxf.comments` module to load comments from DXF files.

The DXF data format is a very versatile and flexible data format and supports various ways to store custom data. This starts by setting special header variables, storing XData, AppData and extension dictionaries in DXF entities and objects, storing XRecords in the OBJECTS section and ends by using proxy entities or even extending the DXF format by user defined entities and objects.

This is the common prolog for all Python code examples shown in this tutorial:

```
import ezdxf

doc = ezdxf.new()
msp = doc.modelspace()
```

Retrieving User Data

Retrieving the is a simple task by *ezdxf*, but often not possible in CAD applications without using the scripting features (AutoLISP) or even the SDK.

AutoLISP Resources

- Autodesk Developer Documentation
- AfraLISP
- Lee Mac Programming

Warning: I have no experience with AutoLISP so far and I created this scripts for AutoLISP while writing this tutorial. There may be better ways to accomplish these tasks, and feedback on this is very welcome. Everything is tested with BricsCAD and should also work with the full version of AutoCAD.

Header Section

The HEADER section has tow ways to store custom data.

Predefined User Variables

There are ten predefined user variables, five 16-bit integer variables called `$USERI1` up to `$USERI5` and five floating point variables (reals) called `$USERR1` up to `$USERR5`. This is very limited and the data maybe will be overwritten by the next application which opens and saves the DXF file. Advantage of this methods is, it works for all supported DXF versions starting at R12.

Settings the data:

```
doc.header["$USERI1"] = 4711
doc.header["$USERR1"] = 3.141592
```

Getting the data by *ezdxf*:

```
i1 = doc.header["$USERI1"]
r1 = doc.header["$USERR1"]
```

Getting the data in *BricsCAD* at the command line:

```
: USERI1
New current value for USERI1 (-32768 to 32767) <4711>:
```

Getting the data by AutoLISP:

```
: (getvar 'USERI1)
4711
```

Setting the value by AutoLISP:

```
: (setvar 'USERI1 1234)
1234
```

Custom Document Properties

This method defines custom document properties, but requires at least DXF R2004. The custom document properties are stored in a *CustomVars* instance in the *custom_vars* attribute of the *HeaderSection* object and supports only string values.

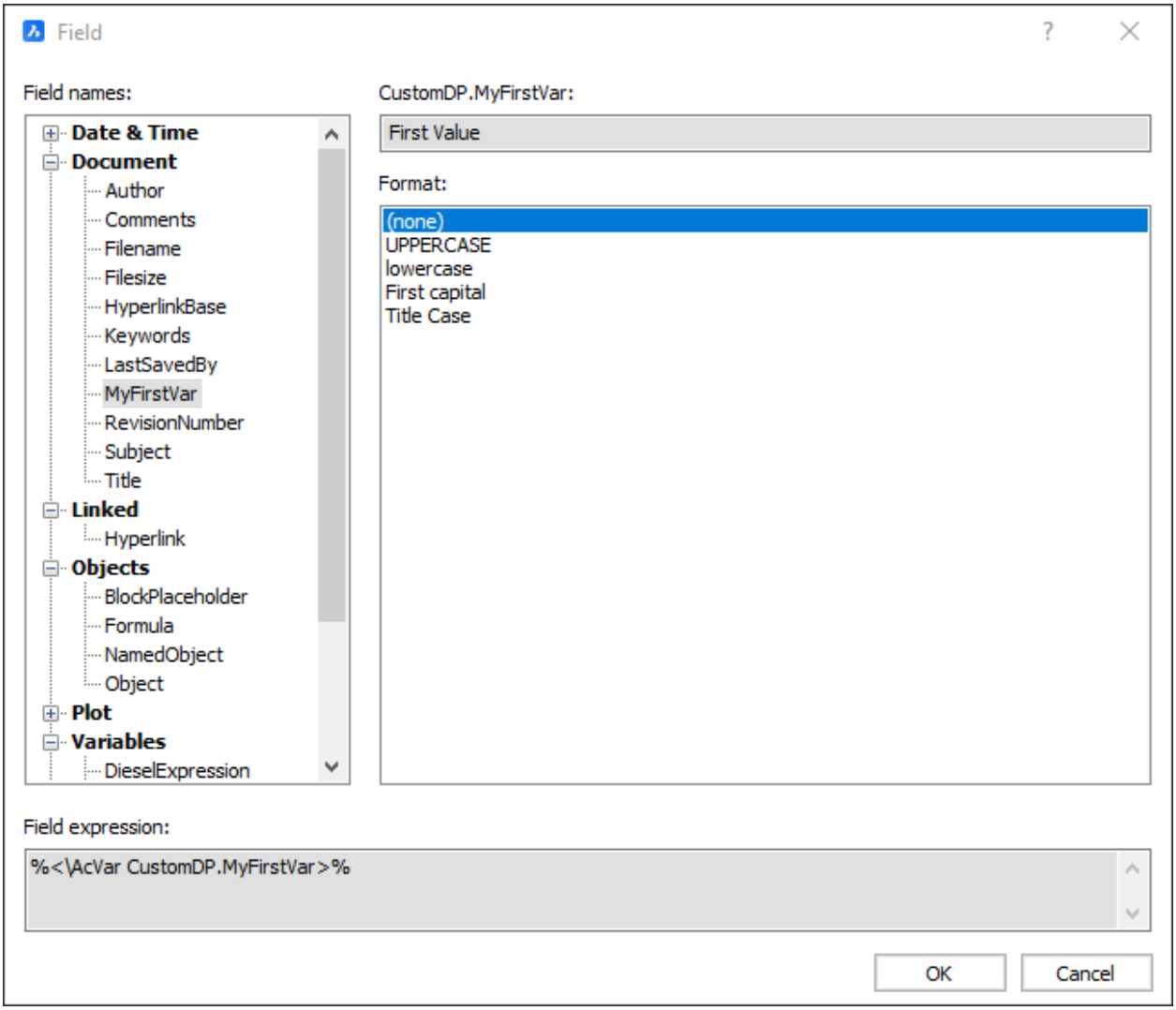
Settings the data:

```
doc.header.custom_vars.append("MyFirstVar", "First Value")
```

Getting the data by *ezdxf*:

```
my_first_var = doc.header.custom_vars.get("MyFirstVar", "Default Value")
```

The document property `MyFirstVar` is available in *BricsCAD* as FIELD variable:



AutoLISP script for getting the custom document properties:

```
(defun C:CUSTOMDOCPROPS (/ Info Num Index Custom)
  (vl-load-com)
  (setq acadObject (vlax-get-acad-object))
  (setq acadDocument (vla-get-ActiveDocument acadObject))

  ;;Get the SummaryInfo
  (setq Info (vlax-get-Property acadDocument 'SummaryInfo))
  (setq Num (vla-NumCustomInfo Info))
  (setq Index 0)
  (repeat Num
    (vla-getCustomByIndex Info Index 'ID 'Value)
    (setq Custom (cons (cons ID Value) Custom))
    (setq Index (1+ Index))
  )  ;repeat

  (if Custom (reverse Custom))
)
```

Running the script in BricsCAD:

```
: (load "customdocprops.lsp")
C:CUSTOMDOCPROPS
: CUSTOMDOCPROPS
(("MyFirstVar" . "First Value"))
```

Meta Data

Starting with version v0.16.4 *ezdxf* stores some meta data in the DXF file and the AppID `EZDXF` will be created. Two entries will be added to the *MetaData* instance, the `CREATED_BY_EZDXF` for DXF documents created by *ezdxf* and the entry `WRITTEN_BY_EZDXF` if the DXF document will be saved by *ezdxf*. The marker string looks like this `"0.17b0 @ 2021-09-18T05:14:37.921826+00:00"` and contains the *ezdxf* version and an UTC timestamp in ISO format.

You can add your own data to the *MetaData* instance as a string with a maximum of 254 characters and choose a good name which may never be used by *ezdxf* in the future.

```
metadata = doc.ezdxf_metadata()
metadata["MY_UNIQUE_KEY"] = "my additional meta data"

print(metadata.get("CREATED_BY_EZDXF"))
print(metadata.get("MY_UNIQUE_KEY"))
```

The data is stored as XDATA in then BLOCK entity of the model space for DXF R12 and for DXF R2000 and later as a DXF *Dictionary* in the root dictionary by the key `EZDXF_META`. See following chapters for accessing such data by AutoLISP.

XDATA

Extended Data (XDATA) is a way to attach arbitrary data to DXF entities. Each application needs a unique AppID registered in the AppID table to add XDATA to an entity. The AppID `ACAD` is reserved and by using *ezdxf* the AppID `EZDXF` is also registered automatically. The total size of XDATA for a single DXF entity is limited to 16kB for AutoCAD. XDATA is supported by all DXF versions and is accessible by AutoLISP.

The valid group codes for extended data are limited to the following values, see also the internals of *Extended Data*:

Group Code	Description
1000	Strings up to 255 bytes long
1001	(fixed) Registered application name up to 31 bytes long
1002	(fixed) An extended data control string `'{'` or `'}'`
1004	Binary data
1005	Database Handle of entities in the drawing database
1010	3D point, in the order X, Y, Z that will not be modified at any transformation of the entity
1011	A WCS point that is moved, scaled, rotated and mirrored along with the entity
1012	A WCS displacement that is scaled, rotated and mirrored along with the entity, but not moved
1013	A WCS direction that is rotated and mirrored along with the entity but not moved and scaled.
1040	A real value
1041	Distance, a real value that is scaled along with the entity
1042	Scale Factor, a real value that is scaled along with the entity
1070	A 16-bit integer (signed or unsigned)
1071	A 32-bit signed (long) integer

Group codes are not unique in the XDATA section and can be repeated, therefore tag order matters.

```
# register your appid
APPID = "YOUR_UNIQUE_ID"
doc.appids.add(APPID)

# create a DXF entity
line = msp.add_line((0, 0), (1, 0))

# setting the data
line.set_xdata(APPID, [
    # basic types
    (1000, "custom text"),
    (1040, 3.141592),
    (1070, 4711),  # 16bit
    (1071, 1_048_576),  # 32bit
    # points and vectors
    (1010, (10, 20, 30)),
    (1011, (11, 21, 31)),
    (1012, (12, 22, 32)),
    (1013, (13, 23, 33)),
    # scaled distances and factors
    (1041, 10),
    (1042, 10),
])

# getting the data
if line.has_xdata(APPID):
    tags = line.get_xdata(APPID)
    print(f"{str(line)} has {len(tags)} tags of XDATA for AppID {APPID!r}")
    for tag in tags:
        print(tag)
```

AutoLISP script for getting XDATA for AppID `YOUR_UNIQUE_ID`:

```
(defun C:SHOWXDATA (/ entity_list xdata_list)
    (setq entity_list (entget (car (entsel)) '("YOUR_UNIQUE_ID")))
    (setq xdata_list (assoc -3 entity_list))
    (car (cdr xdata_list))
)
```

Script output:

```
: SHOWXDATA
Select entity: ("YOUR_UNIQUE_ID" (1000 . "custom text") (1040 . 3.141592) ...
```

See also:

- AfraLISP XDATA tutorial
- *Extended Data (XDATA)* Reference

XDATA Helper Classes

The *XDataUserList* and *XDataUserDict* are helper classes to manage XDATA content in a simple way.

Both classes store the Python types `int`, `float` and `str` and the *ezdxf* type *Vec3*. As the names suggests has the `XDataUserList` a list-like interface and the `XDataUserDict` a dict-like interface. This classes can not contain additional container types, but multiple lists and/or dicts can be stored in the same XDATA section for the same AppID.

These helper classes uses a fixed group code for each data type:

1001	strings (max. 255 chars)
1040	floats
1071	32-bit ints
1010	Vec3

Additional required imports for these examples:

```
from ezdxf.math import Vec3
from ezdxf.entities.xdata import XDataUserDict, XDataUserList
```

Example for *XDataUserDict*:

Each `XDataUserDict` has a unique name, the default name is "DefaultDict" and the default AppID is `EZDXF`. If you use your own AppID, don't forget to create the requited AppID table entry like `doc.appids.new("MyAppID")`, otherwise AutoCAD will not open the DXF file.

```
doc = ezdxf.new()
msp = doc.modelspace()
line = msp.add_line((0, 0), (1, 0))

with XDataUserDict.entity(line) as user_dict:
    user_dict["CreatedBy"] = "mozman"
    user_dict["Float"] = 3.1415
    user_dict["Int"] = 4711
    user_dict["Point"] = Vec3(1, 2, 3)
```

If you modify the content of without using the context manager *entity()*, you have to call *commit()* by yourself, to transfer the modified data back into the XDATA section.

Getting the data back from an entity:

```
with XDataUserDict.entity(line) as user_dict:
    print(user_dict)
    # acts like any other dict()
    storage = dict(user_dict)
```

Example for *XDataUserList*:

This example stores the data in a `XDataUserList` named "AppendedPoints", the default name is "DefaultList" and the default AppID is `EZDXF`.

```
with XDataUserList.entity(line, name="AppendedPoints") as user_list:
    user_list.append(Vec3(1, 0, 0))
    user_list.append(Vec3(0, 1, 0))
    user_list.append(Vec3(0, 0, 1))
```

Now the content of both classes are stored in the same XDATA section for AppID `EZDXF`. The `XDataUserDict` is stored by the name "DefaultDict" and the `XDataUserList` is stored by the name "AppendedPoints".

Getting the data back from an entity:

```
with XDataUserList.entity(line, name="AppendedPoints") as user_list:
    print(user_list)
    storage = list(user_list)

print(f"Copy of XDataUserList: {storage}")
```

See also:

- *XDataUserList* class
- *XDataUserDict* class

Extension Dictionaries

Extension dictionaries are another way to attach custom data to any DXF entity. This method requires DXF R13/14 or later. I will use the short term XDICT for extension dictionaries in this tutorial.

The *Extension Dictionary* is a regular DXF *Dictionary* which can store (key, value) pairs where the key is a string and the value is a DXF object from the OBJECTS section. The usual objects to store custom data are *DictionaryVar* to store simple strings and *XRecord* to store complex data.

Unlike XDATA, custom data attached by extension dictionary will not be transformed along with the DXF entity!

This example shows how to manage the XDICT and to store simple strings as *DictionaryVar* objects in the XDICT, to store more complex data go to the next section *XRecord*.

1. Get or create the XDICT for an entity:

```
# create a DXF entity
line = msp.add_line((0, 0), (1, 0))

if line.has_extension_dict:
    # get the extension dictionary
    xdict = line.get_extension_dict()
else:
    # create a new extension dictionary
    xdict = line.new_extension_dict()
```

2. Add strings as *DictionaryVar* objects to the XDICT. No AppIDs required, but existing keys will be overridden, so be careful by choosing your keys:

```
xdict.add_dictionary_var("DATA1", "Your custom data string 1")
xdict.add_dictionary_var("DATA2", "Your custom data string 2")
```

3. Retrieve the strings from the XDICT as *DictionaryVar* objects:

```
print(f"DATA1 is '{xdict['DATA1'].value}'")
print(f"DATA2 is '{xdict['DATA2'].value}'")
```

The AutoLISP access to DICTIONARIES is possible, but it gets complex and I'm only referring to the AfraLISP Dictionaries and XRecords tutorial.

See also:

- AfraLISP Dictionaries and XRecords Tutorial
- *Extension Dictionary* Reference

- DXF *Dictionary* Reference
- *DictionaryVar* Reference

XRecord

The *XRecord* object can store arbitrary data like the XDATA section, but is not limited by size and can use all group codes in the range from 1 to 369 for *DXF Tags*. The *XRecord* can be referenced by any DXF *Dictionary*, other XRecord objects (tricky ownership!), the XDATA section (store handle by group code 1005) or any other DXF object by adding the XRecord object to the *Extension Dictionary* of the DXF entity.

It is recommend to follow the DXF reference to assign appropriate group codes to *DXF Tags*. My recommendation is shown in the table below, but all group codes from 1 to 369 are valid. I advice against using the group codes 100 and 102 (structure tags) to avoid confusing generic tag loaders. Unfortunately, Autodesk doesn't like general rules and uses DXF format exceptions everywhere.

1	strings (max. 2049 chars)
2	structure tags as strings like "{" and "}"
10	points and vectors
40	floats
90	integers
330	handles

Group codes are not unique in XRecord and can be repeated, therefore tag order matters.

This example shows how to attach a *XRecord* object to a LINE entity by *Extension Dictionary*:

```
line = msp.add_line((0, 0), (1, 0))
line2 = msp.add_line((0, 2), (1, 2))

if line.has_extension_dict:
    xdict = line.get_extension_dict()
else:
    xdict = line.new_extension_dict()

xrecord = xdict.add_xrecord("DATA1")
xrecord.reset([
    (1, "text1"),  # string
    (40, 3.141592),  # float
    (90, 256),  # 32-bit int
    (10, (1, 2, 0)),  # points and vectors
    (330, line2.dxf.handle)  # handles
])

print(xrecord.tags)
```

Script output:

```
[DXFTag(1, 'text1'),
 DXFTag(40, 3.141592),
 DXFTag(90, 256),
 DXFVertex(10, (1.0, 2.0, 0.0)),
 DXFTag(330, '30')]
```

Unlike XDATA, custom data attached by extension dictionary will not be transformed along with the DXF entity! To react to entity modifications by a CAD applications it is possible to write event handlers by AutoLISP, see the AfraLISP Reactors Tutorial for more information. This is very advanced stuff!

See also:

- AfraLISP Dictionaries and XRecords Tutorial
- AfraLISP Reactors Tutorial
- *XRecord* Reference
- helper functions: *ezdxf.lldxf.types.dxftag()* and *ezdxf.lldxf.types.tuples_to_tags()*

XRecord Helper Classes

The *UserRecord* and *BinaryRecord* are helper classes to manage XRECORD content in a simple way. The *UserRecord* manages the data as plain Python types: `dict`, `list`, `int`, `float`, `str` and the *ezdxf* types *Vec2* and *Vec3*. The top level type for the `UserRecord.data` attribute has to be a `list`. The *BinaryRecord* stores arbitrary binary data as BLOB. These helper classes uses fixed group codes to manage the data in XRECORD, you have no choice to change them.

Additional required imports for these examples:

```
from pprint import pprint
import ezdxf
from ezdxf.math import Vec3
from ezdxf.urecord import UserRecord, BinaryRecord
from ezdxf.entities import XRecord
import zlib
```

Example 1: Store entity specific data in the *Extension Dictionary*:

```
line = msp.add_line((0, 0), (1, 0))
xdict = line.new_extension_dict()
xrecord = xdict.add_xrecord("MyData")

with UserRecord(xrecord) as user_record:
    user_record.data = [  # top level has to be a list!
        "MyString",
        4711,
        3.1415,
        Vec3(1, 2, 3),
        {
            "MyIntList": [1, 2, 3],
            "MyFloatList": [4.5, 5.6, 7.8],
        },
    ]
```

Example 1: Get entity specific data back from the *Extension Dictionary*:

```
if line.has_extension_dict:
    xdict = line.get_extension_dict()
    xrecord = xdict.get("MyData")
    if isinstance(xrecord, XRecord):
        user_record = UserRecord(xrecord)
        pprint(user_record.data)
```

If you modify the content of `UserRecord.data` without using the context manager, you have to call *commit()* by yourself, to store the modified data back into the XRECORD.

Example 2: Store arbitrary data in DICTIONARY objects. The XRECORD is stored in the named DICTIONARY, called `rootdict` in *ezdxf*. This DICTIONARY is the root entity for the tree-like data structure stored in the OBJECTS section, see also the documentation of the *ezdxf.sections.objects* module.

```
# Get the existing DICTIONARY object or create a new DICTIONARY object:
my_dict = doc.objects.rootdict.get_required_dict("MyDict")

# Create a new XRECORD object, the DICTIONARY object is the owner of this
# new XRECORD:
xrecord = my_dict.add_xrecord("MyData")

# This example creates the user record without the context manager.
user_record = UserRecord(xrecord)

# Store user data:
user_record.data = [
    "Just another user record",
    4711,
    3.1415,
]
# Store user data in associated XRECORD:
user_record.commit()
```

Example 2: Get user data back from the DICTIONARY object

```
my_dict = doc.rootdict.get_required_dict("MyDict")
entity = my_dict["MyData"]
if isinstance(entity, XRecord):
    user_record = UserRecord(entity)
    pprint(user_record.data)
```

Example 3: Store arbitrary binary data

```
my_dict = doc.rootdict.get_required_dict("MyDict")
xrecord = my_dict.add_xrecord("MyBinaryData")
with BinaryRecord(xrecord) as binary_record:
    # The content is stored as hex strings (e.g. ABBAFEFE...) in one or more
    # group code 310 tags.
    # A preceding group code 160 tag stores the data size in bytes.
    data = b"Store any binary data, even line breaks\r\n" * 20
    # compress data if required
    binary_record.data = zlib.compress(data, level=9)
```

Example 3: Get binary data back from the DICTIONARY object

```
entity = my_dict["MyBinaryData"]
if isinstance(entity, XRecord):
    binary_record = BinaryRecord(entity)
    print("\ncompressed data:")
    pprint(binary_record.data)

    print("\nuncompressed data:")
    pprint(zlib.decompress(binary_record.data))
```

Hint: Don't be fooled, the ability to save any binary data such as images, office documents, etc. in the DXF file doesn't

impress AutoCAD, it simply ignores this data, this data only has a meaning for your application!

See also:

- *urecord* module
- *UserRecord* class
- *BinaryRecord* class

AppData

Application-Defined Data (AppData) was introduced in DXF R13/14 and is used by AutoCAD internally to store the handle to the *Extension Dictionary* and the *Reactors* in DXF entities. *Ezdxf* supports these kind of data storage for any AppID and the data is preserved by AutoCAD and BricsCAD, but I haven't found a way to access this data by AutoLISP or even the SDK. So I don't recommend this feature to store application defined data, because *Extended Data (XDATA)* and the *Extension Dictionary* are well documented and safe ways to attach custom data to entities.

```
# register your appid
APPID = "YOUR_UNIQUE_ID"
doc.appids.add(APPID)

# create a DXF entity
line = msp.add_line((0, 0), (1, 0))

# setting the data
line.set_app_data(APPID, [(300, "custom text"), (370, 4711), (460, 3.141592)])

# getting the data
if line.has_app_data(APPID):
    tags = line.get_app_data(APPID)
    print(f"{str(line)} has {len(tags)} tags of AppData for AppID {APPID!r}")
    for tag in tags:
        print(tag)
```

Printed output:

```
LINE(#30) has 3 tags of AppData for AppID 'YOUR_UNIQUE_ID'
(300, 'custom text')
(370, 4711)
(460, 3.141592)
```

6.6 External References (XREF)

New in version 1.1.

Attached XREFs are links to the modelspace of a specified drawing file. Changes made to the referenced drawing are automatically reflected in the current drawing when it's opened or if the XREF is reloaded.

XREFs can be nested within other XREFs: that is, you can attach an XREF that contains another XREF. You can attach as many copies of an XREF as you want, and each copy can have a different position, scale, and rotation.

You can also overlay an XREF on your drawing. Unlike an attached XREF, an overlaid XREF is not included when the drawing is itself attached or overlaid as an XREF to another drawing.

6.6.1 DXF Files as Attached XREFs

AutoCAD can only display DWG files as attached XREFs, which is a problem since *ezdxf* can only create DXF files. Consequently, any DXF file attached as XREF to a DXF document has be converted to DWG in order to be viewed in AutoCAD.

Fortunately, other CAD applications are more cooperative, so BricsCAD has no problem displaying DXF files as XREFs, although it is not possible to attach a DXF file as an XREF in the BricsCAD application itself.

The *ezdxf.xref* module provides an interface for working with XREFs.

- *attach()* - attach a DXF/DWG file as XREF
- *detach()* - detach a BLOCK definition as XREF
- *embed()* - embed an XREF as a BLOCK definition
- *dxf_info()* - scans a DXF file for basic settings and properties

For loading the content of DWG files is a loading function required, which loads the DWG file as `Drawing` document. The *odafc* add-on module provides such a function: *readfile()*

6.6.2 Importing Data and Resources

The *ezdxf.xref* module replaces the *Importer* add-on.

The basic functionality of the *ezdxf.xref* module is loading data from external files including their required resources, which is an often requested feature by users for importing data from other DXF files into the current document.

The `Importer` add-on was very limited and removed many resources, where the *ezdxf.xref* module tries to preserve as much information as possible.

- *load_modelspace()* - loads the modelspace content from another DXF document
- *load_paperspace()* - loads a paperspace layout from another DXF document
- *write_block()* - writes entities into the modelspace of a new DXF document
- *Loader* - low level loading interface

6.6.3 High Level Functions

`ezdxf.xref.`**`attach`**(*doc:* Drawing, *, *block_name: str*, *filename: str*, *insert:* UVec *= (0, 0, 0)*, *scale: float = 1.0*, *rotation: float = 0.0*, *overlay=False*) → Insert

Attach the file *filename* to the host document as external reference (XREF) and creates a default block reference for the XREF in the modelspace of the document. The function raises a `ValueError` exception if the block definition already exist, but an XREF can be inserted multiple times by adding additional block references:

```
msp.add_blockref(block_name, insert=another_location)
```

Important: If the XREF has different drawing units than the host document, the scale factor between these units must be applied as a uniform scale factor to the block reference! Unfortunately the XREF drawing units can only be detected by scanning the HEADER section of a document by the function *dxf_info()* and is therefore not done automatically by this function. Advice: always use the same units for all drawings of a project!

Parameters

- **doc** – host DXF document
- **block_name** – name of the XREF definition block
- **filename** – file name of the XREF
- **insert** – location of the default block reference
- **scale** – uniform scaling factor
- **rotation** – rotation angle in degrees
- **overlay** – creates an XREF overlay if `True` and an XREF attachment otherwise

Returns
default block reference for the XREF

Return type
Insert

Raises
XrefDefinitionError – block with same name exist

New in version 1.1.

`ezdxf.xref.define`(*doc: Drawing*, *block_name: str*, *filename: str*, *overlay=False*) → None

Add an external reference (xref) definition to a document.

XREF attachment types:

- attached: the XREF that's inserted into this drawing is also present in a document to which this document is inserted as an XREF.
- overlay: the XREF that's inserted into this document is **not** present in a document to which this document is inserted as an XREF.

Parameters

- **doc** – host document
- **block_name** – name of the xref block
- **filename** – external reference filename
- **overlay** – creates an XREF overlay if `True` and an XREF attachment otherwise

Raises
XrefDefinitionError – block with same name exist

New in version 1.1.

`ezdxf.xref.detach`(*block: BlockLayout*, *, *xref_filename: str*) → *Drawing*

Write the content of *block* into the modelspace of a new DXF document and convert *block* to an external reference (XREF). The new DXF document has to be written by the caller: `xref_doc.saveas(xref_filename)`. This way it is possible to convert the DXF document to DWG by the *odafc* add-on if necessary:

```
xref_doc = xref.detach(my_block, "my_block.dwg")
odafc.export_dwg(xref_doc, "my_block.dwg")
```

Parameters

- **block** – block definition to detach
- **xref_filename** – name of the external referenced file

New in version 1.1.

ezdxf.xref.**dxf_info**(*filename: str | PathLike*) → DXFInfo

Scans the HEADER section of a DXF document and returns a `DXFInfo` object, which contains information about the DXF version, text encoding, drawing units and insertion base point.

Raises

IOError – not a DXF file or a generic IO error

ezdxf.xref.**embed**(*xref: BlockLayout*, *, *load_fn: Callable[[str], Drawing] | None = None*, *search_paths: Iterable[Path | str] = tuple()*, *conflict_policy=ConflictPolicy.XREF_PREFIX*) → None

Loads the modelspace of the XREF as content into a block layout.

The loader function loads the XREF as *Drawing* object, by default the function *ezdxf.readfile()* is used to load DXF files. To load DWG files use the *readfile()* function from the *ezdxf.addons.odafc* add-on. The *ezdxf.recover.readfile()* function is very robust for reading DXF files with errors.

If the XREF path isn't absolute the XREF is searched in the folder of the host DXF document and in the *search_path* folders.

Parameters

- **xref** – `BlockLayout` of the XREF document
- **load_fn** – function to load the content of the XREF as *Drawing* object
- **search_paths** – list of folders to search for XREFS, default is the folder of the host document or the current directory if no filepath is set
- **conflict_policy** – how to resolve name conflicts

Raises

- **XrefDefinitionError** – argument *xref* is not a XREF definition
- **FileNotFoundError** – XREF file not found
- ***DXFVersionError*** – cannot load a XREF with a newer DXF version than the host document, try the *odafc* add-on to downgrade the XREF document or upgrade the host document

New in version 1.1.

ezdxf.xref.**load_modelspace**(*sdoc: Drawing*, *tdoc: Drawing*, *filter_fn: Callable[[DXFEntity], bool] | None = None*, *conflict_policy=ConflictPolicy.KEEP*) → None

Loads the modelspace content of the source document into the modelspace of the target document. The filter function *filter_fn* gets every source entity as input and returns `True` to load the entity or `False` otherwise.

Parameters

- **sdoc** – source document
- **tdoc** – target document
- **filter_fn** – optional function to filter entities from the source modelspace
- **conflict_policy** – how to resolve name conflicts

ezdxf.xref.**load_paperspace**(*psp: Paperspace*, *tdoc: Drawing*, *filter_fn: Callable[[DXFEntity], bool] | None = None*, *conflict_policy=ConflictPolicy.KEEP*) → None

Loads the paperspace layout *psp* into the target document. The filter function *filter_fn* gets every source entity as input and returns `True` to load the entity or `False` otherwise.

Parameters

- **psp** – paperspace layout to load

- **tdoc** – target document
- **filter_fn** – optional function to filter entities from the source paperspace layout
- **conflict_policy** – how to resolve name conflicts

ezdxf.xref.**write_block**(*entities: Sequence[DXFEntity]*, *, *origin:* UVec = *(0, 0, 0)*) → *Drawing*

Write *entities* into the modelspace of a new DXF document.

This function is called "write_block" because the new DXF document can be used as an external referenced block. This function is similar to the WBLOCK command in CAD applications.

Virtual entities are not supported, because each entity needs a real database- and owner handle.

Parameters

- **entities** – DXF entities to write
- **origin** – block origin, defines the point in the modelspace which will be inserted at the insert location of the block reference

Raises

EntityError – virtual entities are not supported

New in version 1.1.

6.6.4 Conflict Policy

class ezdxf.xref.**ConflictPolicy**

KEEP

XREF_PREFIX

NUM_PREFIX

6.6.5 Low Level Loading Interface

class ezdxf.xref.**Loader**(*sdoc:* Drawing, *tdoc:* Drawing, *conflict_policy=ConflictPolicy.KEEP*)

Load entities and resources from the source DXF document *sdoc* into a target DXF document.

6.7 Howto

The Howto section show how to accomplish specific tasks with *ezdxf* in a straight forward way without teaching basics or internals, if you are looking for more information about the ezdxf internals look at the *Reference* section or if you want to learn how to use *ezdxf* go to the *Tutorials* section or to the *Basic Concepts* section.

6.7.1 General Document

General preconditions:

```
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f"Not a DXF file or a generic I/O error.")
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f"Invalid or corrupted DXF file.")
    sys.exit(2)
msp = doc.modelspace()
```

This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the *ezdxf.recover* module.

Load DXF Files with Structure Errors

If you know the files you will process have most likely minor or major flaws, use the *ezdxf.recover* module:

```
import sys
from ezdxf import recover

try:  # low level structure repair:
    doc, auditor = recover.readfile(name)
except IOError:
    print(f"Not a DXF file or a generic I/O error.")
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f"Invalid or corrupted DXF file: {name}.")
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f"Found unrecoverable errors in DXF file: {name}.")
    auditor.print_error_report()
```

For more loading scenarios follow the link: *ezdxf.recover*

Set/Get Header Variables

ezdxf has an interface to get and set HEADER variables:

```
doc.header["VarName"] = value
value = doc.header["VarName"]
```

See also:

`HeaderSection` and online documentation from Autodesk for available header variables.

Set DXF Drawing Units

The header variable $INSUNITS defines the drawing units for the modelspace and therefore for the DXF document if no further settings are applied. The most common units are 6 for meters and 1 for inches.

Use this HEADER variables to setup the default units for CAD applications opening the DXF file. This setting is not relevant for *ezdxf* API calls, which are unitless for length values and coordinates and decimal degrees for angles (in most cases).

Sets drawing units:

```
doc.header["$INSUNITS"] = 6
```

For more information see section *DXF Units*.

Create More Readable DXF Files (DXF Pretty Printer)

DXF files are plain text files, you can open this files with every text editor which handles bigger files. But it is not really easy to get quick the information you want.

Create a more readable HTML file (DXF Pretty Printer):

```
# Call as executable script from the command line:
ezdxf pp FILE [FILE ...]

# Call as module on Windows:
py -m ezdxf pp FILE [FILE ...]

# Call as module on Linux/Mac
python3 -m ezdxf pp FILE [FILE ...]
```

This creates a HTML file with a nicer layout than a plain text file, and handles are links between DXF entities, this simplifies the navigation between the DXF entities.

```
usage: ezdxf pp [-h] [-o] [-r] [-x] [-l] FILE [FILE ...]

positional arguments:
  FILE             DXF files pretty print

optional arguments:
  -h, --help       show this help message and exit
  -o, --open       open generated HTML file with the default web browser
  -r, --raw        raw mode - just print tags, no DXF structure interpretation
  -x, --nocompile  don't compile points coordinates into single tags (only in
                   raw mode)
  -l, --legacy     legacy mode - reorders DXF point coordinates
```

Important: This does not render the graphical content of the DXF file to a HTML canvas element.

Calculate Extents for the Modelspace

Since *ezdxf* v0.16 exist a *ezdxf.bbox* module to calculate bounding boxes for DXF entities. This module makes the extents calculation very easy, but read the documentation for the *bbox* module to understand its limitations.

```
import ezdxf
from ezdxf import bbox

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()

extents = bbox.extents(msp)
```

The returned *extents* is a *ezdxf.math.BoundingBox* object.

Set Initial View/Zoom for the Modelspace

To show an arbitrary location of the modelspace centered in the CAD application window, set the `'*Active'` VPORT to this location. The DXF attribute `dxf.center` defines the location in the modelspace, and the `dxf.height` specifies the area of the modelspace to view. Shortcut function:

```
doc.set_modelspace_vport(height=10, center=(10, 10))
```

See also:

The *ezdxf.zoom* module is another way to set the initial modelspace view.

Setting the initial view to the extents of all entities in the modelspace:

```
import ezdxf
from ezdxf import zoom

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
zoom.extents(msp)
```

Setting the initial view to the extents of just some entities:

```
lines = msp.query("LINES")
zoom.objects(lines)
```

The *zoom* module also works for paperspace layouts.

Important: The *zoom* module uses the *bbox* module to calculate the bounding boxes for DXF entities. Read the documentation for the *bbox* module to understand its limitations and the bounding box calculation for large documents can take a while!

Hide the UCS Icon

The visibility of the UCS icon is controlled by the DXF *ucs_icon* attribute of the *VPort* entity:

- bit 0: 0=hide, 1=show
- bit 1: 0=display in lower left corner, 1=display at origin

The state of the UCS icon can be set in conjunction with the initial *VPort* of the model space, this code turns off the UCS icon:

```
doc.set_modelspace_vport(10, center=(10, 10), dxfattribs={"ucs_icon": 0})
```

Alternative: turn off UCS icons for all `VPort` entries in the active viewport configuration:

```
for vport in doc.viewports.get_config("*Active"):
    vport.dxf.ucs_icon = 0
```

Show Lineweights in DXF Viewers

By default lines and curves are shown without lineweights in DXF viewers. By setting the header variable $LWDISPLAY to 1 the DXF viewer should display lineweights, if supported by the viewer.

```
doc.header["$LWDISPLAY"] = 1
```

Add *ezdxf* Resources to Existing DXF Document

Add all *ezdxf* specific resources (line types, text- and dimension styles) to an existing DXF document:

```
import ezdxf
from ezdxf.tools.standards import setup_drawing

doc = ezdxf.readfile("your.dxf")
setup_drawing(doc, topics="all")
```

Set Logging Level of *ezdxf*

Set the logging level of the *ezdxf* package to a higher level to minimize logging messages from ezdxf. At level `ERROR` only severe errors will be logged and `WARNING`, `INFO` and `DEBUG` messages will be suppressed:

```
import logging

logging.getLogger("ezdxf").setLevel(logging.ERROR)
```

6.7.2 DXF Viewer

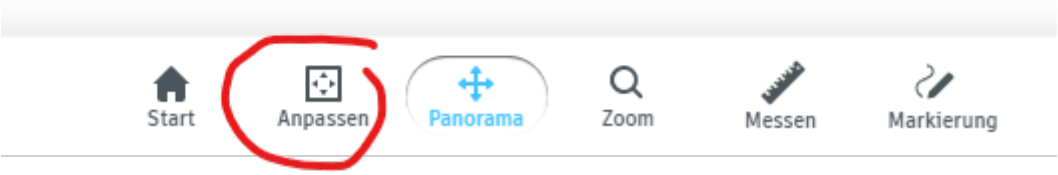
A360 Viewer Problems

AutoDesk web service A360 seems to be more picky than the AutoCAD desktop applications, may be it helps to use the latest DXF version supported by ezdxf, which is DXF R2018 (AC1032) in the year of writing this lines (2018).

DXF Entities Are Not Displayed in the Viewer

ezdxf does not automatically locate the main viewport of the modelspace at the entities, you have to perform the "Zoom to Extends" command, here in TrueView 2020:

And here in the Autodesk Online Viewer:



Add this line to your code to relocate the main viewport, adjust the *center* (in modelspace coordinates) and the *height* (in drawing units) arguments to your needs:

```
doc.set_modelspace_vport(height=10, center=(0, 0))
```

Show IMAGES/XREFS on Loading in AutoCAD

If you are adding XREFS and IMAGES with relative paths to existing drawings and they do not show up in AutoCAD immediately, change the HEADER variable `$PROJECTNAME=''` to *(not really)* solve this problem. The ezdxf templates for DXF R2004 and later have `$PROJECTNAME=''` as default value.

Thanks to David Booth:

> If the filename in the IMAGEDEF contains the full path (absolute in AutoCAD) then it shows on loading, otherwise it won't display (reports as unreadable) until you manually reload using XREF manager.
>
> A workaround (to show IMAGES on loading) appears to be to save the full file path in the DXF or save it as a DWG.

Thanks to Zac Luzader:

> Has anyone else noticed that very short simple image file names seem to avoid this problem? Once I ensured that the image file's name was short and had no special characters (letters, numbers and underscores only) the problem seemed to go away. I didn't rigorously analyze the behavior as its very time consuming.
>
> Also: You can safely put the image in a subdirectory and use a relative path. The name of the subdirectory does not seem to trigger this problem, provided that the image file name itself is very short and simple.
>
> Also pro tip: The XRef manager exists in DWG TrueView 2023, but access to it is only possible if you have a completely broken reference. Create a DXF with a reference to a non-existent file, then the error dialog will let you open the XRef Manager. Once it is open you can pin it and it will be open next time, even if you have no broken references.

See also:

Discussion on github: Images don't show in AutoCAD until …

Set Initial View/Zoom for the Modelspace

See section "General Document": *Set Initial View/Zoom for the Modelspace*

Show Lineweights in DXF Viewers

By default lines and curves are shown without lineweights in DXF viewers. By setting the header variable $LWDISPLAY to 1 the DXF viewer should display lineweights, if supported by the viewer.

```
doc.header["$LWDISPLAY"] = 1
```

6.7.3 DXF Content

General preconditions:

```
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)
msp = doc.modelspace()
```

Get/Set Entity Color

The entity color is stored as *ACI* (AutoCAD Color Index):

```
aci = entity.dxf.color
```

Default value is 256 which means BYLAYER:

```
layer = doc.layers.get(entity.dxf.layer)
aci = layer.get_color()
```

The special *get_color()* method is required, because the color attribute `Layer.dxf.color` is misused as layer on/off flag, a negative color value means the layer is off.

ACI value 0 means BYBLOCK, which means the color from the block reference (INSERT entity).

Set color as ACI value as int in range [0, 256]:

```
entity.dxf.color = 1
```

The ACI value 7 has a special meaning, it is white on dark backgrounds and white on light backgrounds.

Get/Set Entity RGB Color

RGB true color values are supported since DXF R13 (AC1012), the 24-bit RGB value is stored as integer in the DXF attribute *true_color*:

```
# 24 bit binary value: 0bRRRRRRRRGGGGGGGGBBBBBBBB or hex value: 0xRRGGBB
# set true color value to red
entity.dxf.true_color = 0xFF0000
```

Use the helper functions from the *ezdxf.colors* module for RGB integer value handling:

```
from ezdxf import colors

entity.dxf.true_color = colors.rgb2int((0xFF, 0, 0))
r, g, b = colors.int2rgb(entity.dxf.true_color)
```

The RGB values of the AutoCAD default colors are not officially documented, but an accurate translation table is included in *ezdxf*:

```
# Warning: ACI value 256 (BYLAYER) raises an IndexError!
rgb24 = colors.DXF_DEFAULT_COLORS[aci]
print(f"RGB Hex Value: #{rgb24:06X}")
r, g, b = colors.int2rgb(rgb24)
print(f"RGB Channel Values: R={r:02X} G={g:02X} b={b:02X}")
```

If `color` and `true_color` values are set, BricsCAD and AutoCAD use the `true_color` value as display color for the entity.

Get/Set True Color as RGB-Tuple

Get/Set the true color value as (r, g, b)-tuple by the *rgb* property of the *DXFGraphic* entity:

```
# set true color value to red
entity.rgb = (0xFF, 0, 0)

# get true color values
r, g, b = entity.rgb
```

Get/Set Block Reference Attributes

Block references (*Insert*) can have attached attributes (*Attrib*), these are simple text annotations with an associated tag appended to the block reference.

Iterate over all appended attributes:

```
# get all INSERT entities with entity.dxf.name == "Part12"
blockrefs = msp.query('INSERT[name=="Part12"]')
if len(blockrefs):
    entity = blockrefs[0]  # process first entity found
    for attrib in entity.attribs:
        if attrib.dxf.tag == "diameter":  # identify attribute by tag
            attrib.dxf.text = "17mm"  # change attribute content
```

Get attribute by tag:

```
diameter = entity.get_attrib('diameter')
if diameter is not None:
    diameter.dxf.text = "17mm"
```

Adding XDATA to Entities

Adding XDATA as list of tuples (group code, value) by *set_xdata()*, overwrites data if already present:

```
doc.appids.new('YOUR_APPID')  # IMPORTANT: create an APP ID entry

circle = msp.add_circle((10, 10), 100)
circle.set_xdata(
    'YOUR_APPID',
    [
        (1000, 'your_web_link.org'),
        (1002, '{'),
        (1000, 'some text'),
        (1002, '{'),
        (1071, 1),
        (1002, '}'),
        (1002, '}')
    ])
```

For group code meaning see DXF reference section DXF Group Codes in Numerical Order Reference, valid group codes are in the range 1000 - 1071.

Method *get_xdata()* returns the extended data for an entity as *Tags* object.

See also:

Tutorial: *Storing Custom Data in DXF Files*

Get Overridden DIMSTYLE Values from DIMENSION

In general the *Dimension* styling and config attributes are stored in the `Dimstyle` entity, but every attribute can be overridden for each DIMENSION entity individually, get overwritten values by the `DimstyleOverride` object as shown in the following example:

```
for dimension in msp.query('DIMENSION'):
    dimstyle_override = dimension.override()  # requires v0.12
    dimtol = dimstyle_override['dimtol']
    if dimtol:
        print(f'{str(dimension)} has tolerance values:')
        dimtp = dimstyle_override['dimtp']
        dimtm = dimstyle_override['dimtm']
        print(f'Upper tolerance: {dimtp}')
        print(f'Lower tolerance: {dimtm}')
```

The `DimstyleOverride` object returns the value of the underlying DIMSTYLE objects if the value in DIMENSION was not overwritten, or `None` if the value was neither defined in DIMSTYLE nor in DIMENSION.

Override DIMSTYLE Values for DIMENSION

Same as above, the `DimstyleOverride` object supports also overriding DIMSTYLE values. But just overriding this values have no effect on the graphical representation of the DIMENSION entity, because CAD applications just show the associated anonymous block which contains the graphical representation on the DIMENSION entity as simple DXF entities. Call the `render` method of the `DimstyleOverride` object to recreate this graphical representation by *ezdxf*, but *ezdxf* **does not** support all DIMENSION types and DIMVARS yet, and results **will differ** from AutoCAD or BricsCAD renderings.

```
dimstyle_override = dimension.override()
dimstyle_override.set_tolerance(0.1)

# delete associated geometry block
del doc.blocks[dimension.dxf.geometry]

# recreate geometry block
dimstyle_override.render()
```

How to Change the HATCH Pattern Origin Point

This code sets the origin of the first pattern line to the given *origin* and the origins of all remaining pattern lines relative to the first pattern line origin.

```
from ezdxf.entities import Hatch, Pattern
from ezdxf.math import Vec2

def shift_pattern_origin(hatch: Hatch, offset: Vec2):
    if isinstance(hatch.pattern, Pattern):
        for pattern_line in hatch.pattern.lines:
            pattern_line.base_point += offset

def reset_pattern_origin_of_first_pattern_line(hatch: Hatch, origin: Vec2):
    if isinstance(hatch.pattern, Pattern) and len(hatch.pattern.lines):
        first_pattern_line = hatch.pattern.lines[0]
        offset = origin - first_pattern_line.base_point
        shift_pattern_origin(hatch, offset)
```

See also:

- Discussion #769

How to Get the Length of a Spline or Polyline

There exist no analytical function to calculate the length of a B-spline, you have to approximate the curve and calculate the length of the polyline. The construction tool *ezdxf.math.ConstructionPolyline* is may be useful for that.

```
import ezdxf
from ezdxf.math import ConstructionPolyline

doc = ezdxf.new()
msp = doc.modelspace()
fit_points = [(0, 0, 0), (750, 500, 0), (1750, 500, 0), (2250, 1250, 0)]

spline = msp.add_spline(fit_points)
# Adjust the max. sagitta distance to your needs or run the calculation in a loop
```

(continues on next page)

(continued from previous page)

```
# reducing the distance until the difference to the previous run is smaller
# than your expected precision:
polyline = ConstructionPolyline(spline.flattening(distance=0.1))
print(f"approximated length = {polyline.length:.2f}")
```

How to Resolve DXF Properties

Graphical properties of DXF entities (color, lineweight, …) are sometimes hard to resolve because of the complex possibilities to inherit properties from layers or blocks, or overriding them by *ctb* files.

The *drawing* add-on provides the *RenderContext* class that can be used to resolve properties of entities in the context of their use:

```
import ezdxf
from ezdxf.addons.drawing.properties import RenderContext

doc = ezdxf.new()
doc.layers.add("LINE", color=ezdxf.colors.RED)
msp = doc.modelspace()
line = msp.add_line((0, 0), (1, 0), dxfattribs={"layer": "LINE"})

ctx = RenderContext(doc)
ctx.set_current_layout(msp)
print(f"resolved RGB value: {ctx.resolve_color(line)}")
```

Output:

```
resolved RGB value: #ff0000
```

This works in most simple cases, resolving properties of objects in viewports or nested blocks requires additional information that is beyond the scope of a simple guide.

6.7.4 Fonts

Rendering SHX Fonts

The SHX font format is not documented nor supported by many libraries/packages like *Matplotlib* and *Qt*, therefore only SHX fonts which have corresponding TTF-fonts can be rendered by these backends. The mapping from/to SHX/TTF fonts is hard coded in the source code file: ezdxf/tools/fonts.py

Rebuild Internal Font Cache

Ezdxf uses Matplotlib to manage fonts and caches the collected information. If you wanna use new installed fonts which are not included in the default cache files of *ezdxf* you have to rebuild the cache files:

```
import ezdxf
from ezdxf.tools import fonts

# xdg_path() returns "$XDG_CACHE_HOME/ezdxf" or "~/.cache/ezdxf" if
# $XDG_CACHE_HOME is not set
font_cache_dir = ezdxf.options.xdg_path("XDG_CACHE_HOME", ".cache")
fonts.build_system_font_cache(path=font_cache_dir)
```

(continues on next page)

(continued from previous page)

```
ezdxf.options.font_cache_directory = font_cache_dir
# Save changes to the default config file "~/.config/ezdxf/ezdxf.ini"
# to load the font cache always from the new location.
ezdxf.options.write_home_config()
```

For more information see the *ezdxf.options* and the *ezdxf.tools.fonts* module.

Matplotlib Doesn't Find Fonts

If *Matplotlib* does not find an installed font and rebuilding the matplotlib font cache does not help, deleting the cache file `~/.matplotlib/fontlist-v330.json` (or similar file in newer versions) may help.

For more information see the *ezdxf.tools.fonts* module.

6.7.5 Drawing Add-on

This section consolidates the FAQ about the drawing add-on from the github forum.

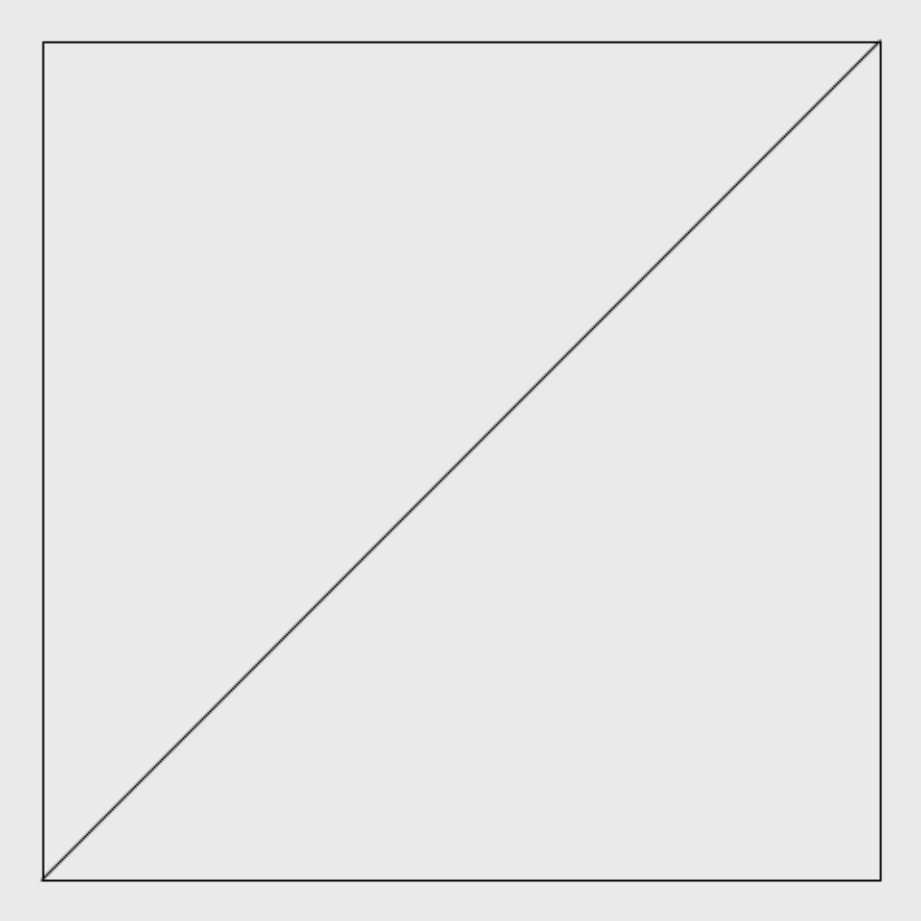
All Backends

How to Set Background and Foreground Colors

Override the default background and foreground colors. The foreground color is the *AutoCAD Color Index (ACI)* 7, which is white/black depending on the background color. If the foreground color is not specified, the foreground color is white for dark backgrounds and black for light backgrounds. The required color format is a hex string "#RRGGBBAA".

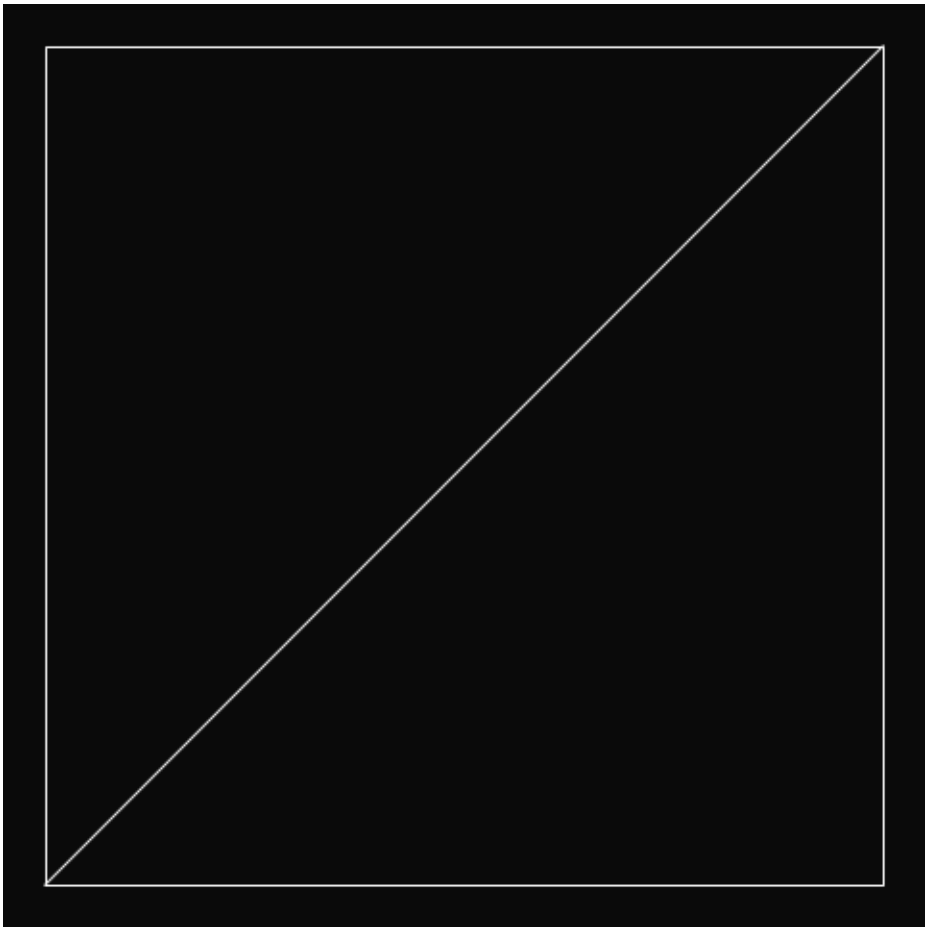
```
from ezdxf.addons.drawing.properties import LayoutProperties

# -x-x-x snip -x-x-x-

fig: plt.Figure = plt.figure()
ax: plt.Axes = fig.add_axes((0, 0, 1, 1))
ctx = RenderContext(doc)

# get the modelspace properties
msp_properties = LayoutProperties.from_layout(msp)

# set light gray background color and black foreground color
msp_properties.set_colors("#eaeaea")
out = MatplotlibBackend(ax)

# override the layout properties and render the modelspace
Frontend(ctx, out).draw_layout(
    msp,
    finalize=True,
    layout_properties=msp_properties,
)
fig.savefig("image.png")
```

A light background "#eaeaea" has a black foreground color by default:



A dark background “#0a0a0a” has a white foreground color by default:

```
# -x-x-x snip -x-x-x-

msp_properties.set_colors("#0a0a0a")

# -x-x-x snip -x-x-x-
```



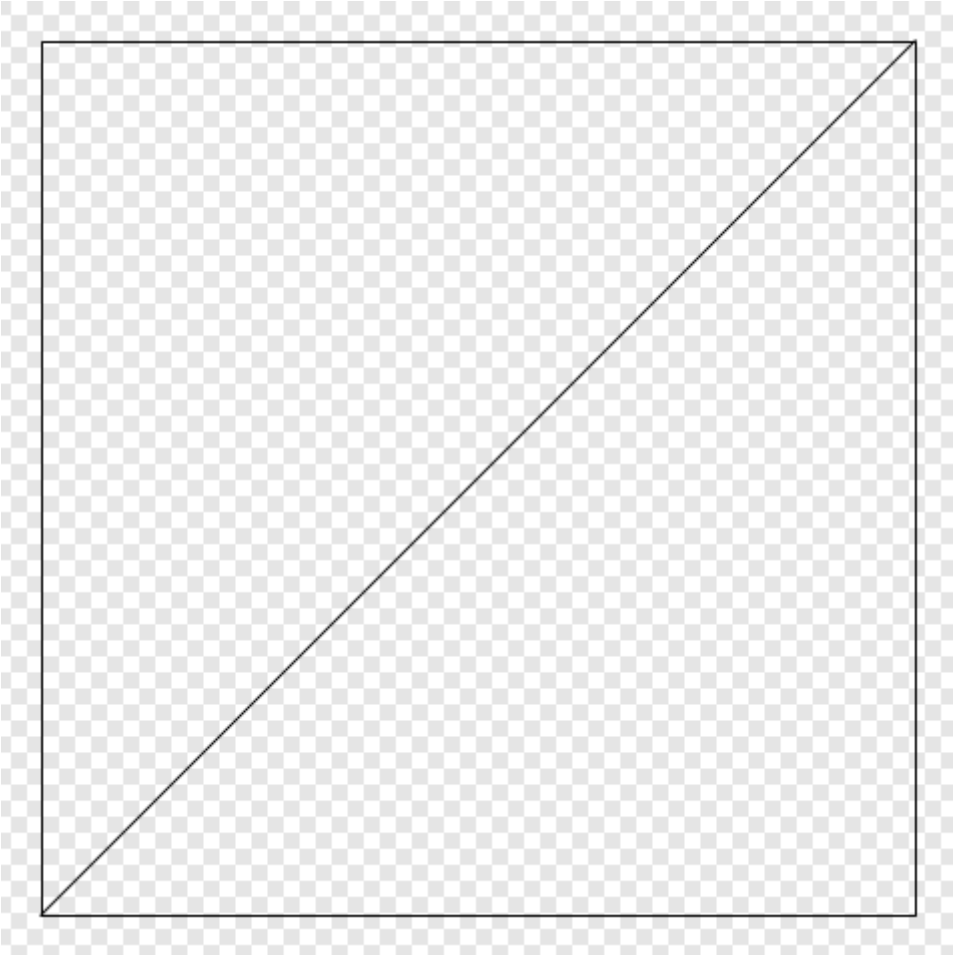
How to Set a Transparent Background Color

The override color include an alpha transparency "#RRGGBBAA" value. An alpha value of "00" is opaque and "ff" is fully transparent. A transparent background color still defines the foreground color!

Hint: The `savefig()` function of the matplotlib backend requires the *transparent* argument to be set to `True` to support transparency.

A light and fully transparent background "#eaeaeaff" has a black foreground color by default:

```
# -x-x-x snip -x-x-x-

msp_properties.set_colors("#eaeaeaff")

# -x-x-x snip -x-x-x-

fig.savefig("image.png", transparent=True)
```



A dark and fully transparent background “#0a0a0aff” has a **white** foreground color by default:

```
# -x-x-x snip -x-x-x-

msp_properties.set_colors("#0a0a0aff")

# -x-x-x snip -x-x-x-

fig.savefig("image.png", transparent=True)
```

How to Exclude DXF Entities from Rendering

- If all unwanted entities are on the same layer switch off the layer.
- If the document is not saved later, you can delete the entities or set them invisible.
- Filter the unwanted entities by a filter function.

The argument *filter_func* of the `Frontend.draw_layout()` method expects a function which takes a graphical DXF entity as input and returns `True` if the entity should be rendered or `False` to exclude the entity from rendering.

This filter function excludes all DXF entities with an ACI color value of 2:

```
from ezdxf.entities import DXFGraphic

def my_filter(e: DXFGraphic) -> bool:
    return e.dxf.color != 2

# -x-x-x snip -x-x-x-

Frontend(ctx, out).draw_layout(msp, finalize=True, filter_func=my_filter)
```

Important: Not all attributes have a default value if the attribute does not exist. If you are not sure about this, use the `get()` method:

```
def my_filter(e: DXFGraphic) -> bool:
    return e.dxf.get("color", 7) != 2
```

How to Override Properties of DXF Entities

Create a custom `Frontend` class and override the the `override_properties()` method:

```
class MyFrontend(Frontend):
    def override_properties(self, entity: DXFGraphic, properties: Properties) -> None:
        # remove alpha channel from all entities, "#RRGGBBAA"
        properties.color = properties.color[:7]

# -x-x-x snip -x-x-x-

MyFrontend(ctx, out).draw_layout(msp, finalize=True)
```

See also:

- *ezdxf.addons.drawing.properties.Properties*

Matplotlib Backend

See also:

- Matplotlib package: https://matplotlib.org/stable/api/matplotlib_configuration_api.html
- `Figure` API: https://matplotlib.org/stable/api/figure_api.html
- `Axes` API: https://matplotlib.org/stable/api/axis_api.html

How to Get the Pixel Coordinates of DXF Entities

See also:

- Source: https://github.com/mozman/ezdxf/discussions/219

Transformation from modelspace coordinates to image coordinates:

```
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw

import ezdxf
from ezdxf.math import Matrix44
from ezdxf.addons.drawing import RenderContext, Frontend
from ezdxf.addons.drawing.matplotlib import MatplotlibBackend

def get_wcs_to_image_transform(
    ax: plt.Axes, image_size: tuple[int, int]
) -> Matrix44:
    """Returns the transformation matrix from modelspace coordinates to image
    coordinates.
    """

    x1, x2 = ax.get_xlim()
```

(continues on next page)

(continued from previous page)

```
    y1, y2 = ax.get_ylim()
    data_width, data_height = x2 - x1, y2 - y1
    image_width, image_height = image_size
    return (
        Matrix44.translate(-x1, -y1, 0)
        @ Matrix44.scale(
            image_width / data_width, -image_height / data_height, 1.0
        )
        # +1 to counteract the effect of the pixels being flipped in y
        @ Matrix44.translate(0, image_height + 1, 0)
    )

# create the DXF document
doc = ezdxf.new()
msp = doc.modelspace()
msp.add_lwpolyline([(0, 0), (1, 0), (1, 1), (0, 1)], close=True)
msp.add_line((0, 0), (1, 1))

# export the pixel image
fig: plt.Figure = plt.figure()
ax: plt.Axes = fig.add_axes([0, 0, 1, 1])
ctx = RenderContext(doc)
out = MatplotlibBackend(ax)
Frontend(ctx, out).draw_layout(msp, finalize=True)
fig.savefig("cad.png")
plt.close(fig)

# reload the pixel image by Pillow (PIL)
img = Image.open("cad.png")
draw = ImageDraw.Draw(img)

# add some annotations to the pixel image by using modelspace coordinates
m = get_wcs_to_image_transform(ax, img.size)
a, b, c = (
    (v.x, v.y)  # draw.line() expects tuple[float, float] as coordinates
    # transform modelspace coordinates to image coordinates
    for v in m.transform_vertices([(0.25, 0.75), (0.75, 0.25), (1, 1)])
)
draw.line([a, b, c, a], fill=(255, 0, 0))

# show the image by the default image viewer
img.show()
```

How to Get Modelspace Coordinates from Pixel Coordinates

This is the reverse operation of the previous how-to: *How to Get the Pixel Coordinates of DXF Entities*

See also:

- Full example script: wcs_to_image_coordinates.py
- Source: https://github.com/mozman/ezdxf/discussions/269

```
def get_image_to_wcs_transform(
    ax: plt.Axes, image_size: tuple[int, int]
) -> Matrix44:
```

(continues on next page)

(continued from previous page)

```
    m = get_wcs_to_image_transform(ax, image_size)
    m.inverse()
    return m

# -x-x-x snip -x-x-x-

img2wcs = get_image_to_wcs_transform(ax, img.size)
print(f"0.25, 0.75 == {img2wcs.transform(a).round(2)}")
print(f"0.75, 0.25 == {img2wcs.transform(b).round(2)}")
print(f"1.00, 1.00 == {img2wcs.transform(c).round(2)}")
```

How to Export a Specific Area of the Modelspace

This code exports the specified modelspace area from (5, 3) to (7, 8) as a 2x5 inch PNG image to maintain the aspect ratio of the source area.

Use case: render only a specific area of the modelspace.

See also:

- Full example script: export_specific_area.py
- Source: https://github.com/mozman/ezdxf/discussions/451

```
# -x-x-x snip -x-x-x-

# export the pixel image
fig: plt.Figure = plt.figure()
ax: plt.Axes = fig.add_axes([0, 0, 1, 1])
ctx = RenderContext(doc)
out = MatplotlibBackend(ax)
Frontend(ctx, out).draw_layout(msp, finalize=True)

# setting the export area:
xmin, xmax = 5, 7
ymin, ymax = 3, 8
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)

# set the output size to get the expected aspect ratio:
fig.set_size_inches(xmax - xmin, ymax - ymin)
fig.savefig("x5y3_to_x7y8.png")
plt.close(fig)
```

How to Render Without Margins

To remove the empty space at the image borders set the margins of the `Axes` object to zero:

```
ax.margins(0)
fig.savefig("image_without_margins.png")
plt.close(fig)
```

See also:

- Matplotlib docs about margins

How to Set the Pixel Count per Drawing Unit

This code exports the modelspace with an extent of 5 x 3 drawing units with 100 pixels per drawing unit as a 500 x 300 pixel image.

Use case: render the content with a fixed number of pixels for a drawing unit, e.g. a drawing unit of 1 inch should be rendered by 100 pixels.

See also:

- Full example script: export_image_pixel_size.py
- Source: https://github.com/mozman/ezdxf/discussions/357

```
# -x-x-x snip -x-x-x-

def set_pixel_density(fig: plt.Figure, ax: plt.Axes, ppu: int):
    """Argument `ppu` is pixels per drawing unit."""
    xmin, xmax = ax.get_xlim()
    width = xmax - xmin
    ymin, ymax = ax.get_ylim()
    height = ymax - ymin
    dpi = fig.dpi
    width_inch = width * ppu / dpi
    height_inch = height * ppu / dpi
    fig.set_size_inches(width_inch, height_inch)

# -x-x-x snip -x-x-x-

# export image with 100 pixels per drawing unit = 500x300 pixels
set_pixel_density(fig, ax, 100)
fig.savefig("box_500x300.png")
plt.close(fig)
```

How to Export a Specific Image Size in Pixels

This code exports the modelspace with an extent of 5 x 3 drawing units as a 1000 x 600 pixel Image.

Use case: render the content with a fixed image size in pixels.

See also:

- Full example script: export_image_pixel_size.py
- Source: https://github.com/mozman/ezdxf/discussions/357

```
# -x-x-x snip -x-x-x-

def set_pixel_size(fig: plt.Figure, size: tuple[int, int]):
    x, y = size
    fig.set_size_inches(x / fig.dpi, y / fig.dpi)

# -x-x-x snip -x-x-x-

# export image with a size of 1000x600 pixels
set_pixel_size(fig, (1000, 600))
fig.savefig("box_1000x600.png")
plt.close(fig)
```

How to Set the Page Size in Inches

The page- or image size in inches is set by the `set_size_inches()` method of the `Figure` class. The content within the `Axes` limits will be scaled to fill the page.

Use case: render the whole content to a PDF document with a specific paper size without worrying about scale.

```
fig.set_size_inches(8, 11)
```

How to Render at a Specific Scale

This code exports the modelspace at a specific scale and paper size.

Use case: render the content to a PDF document with a specific paper size and scale, but not all content may be rendered.

See also:

- Full example script: render_to_scale.py
- Source: https://github.com/mozman/ezdxf/discussions/665

```
# -x-x-x snip -x-x-x-

def render_limits(
    origin: tuple[float, float],
    size_in_inches: tuple[float, float],
    scale: float,
) -> tuple[float, float, float, float]:
    """Returns the final render limits in drawing units.

    Args:
         origin: lower left corner of the modelspace area to render
         size_in_inches: paper size in inches
         scale: render scale, e.g. scale=100 means 1:100, 1m is
             rendered as 0.01m or 1cm on paper

    """
    min_x, min_y = origin
    max_x = min_x + size_in_inches[0] * scale
    max_y = min_y + size_in_inches[1] * scale
    return min_x, min_y, max_x, max_y


def export_to_scale(
    paper_size: tuple[float, float] = (8.5, 11),
    origin: tuple[float, float] = (0, 0),
    scale: float = 1,
    dpi: int = 300,
):
    """Render the modelspace content with to a specific paper size and scale.

    Args:
        paper_size: paper size in inches
        origin: lower left corner of the modelspace area to render
        scale: render scale, e.g. scale=100 means 1:100, 1m is
            rendered as 0.01m or 1cm on paper
        dpi: pixel density on paper as dots per inch
```

(continues on next page)

(continued from previous page)

```
    """
    # -x-x-x snip -x-x-x-

    ctx = RenderContext(doc)
    fig: plt.Figure = plt.figure(dpi=dpi)
    ax: plt.Axes = fig.add_axes([0, 0, 1, 1])

    # disable all margins
    ax.margins(0)

    # get the final render limits in drawing units:
    min_x, min_y, max_x, max_y = render_limits(
        origin, paper_size, scale
    )

    ax.set_xlim(min_x, max_x)
    ax.set_ylim(min_y, max_y)

    out = MatplotlibBackend(ax)
    # finalizing invokes auto-scaling by default!
    Frontend(ctx, out).draw_layout(msp, finalize=False)

    # set output size in inches:
    fig.set_size_inches(paper_size[0], paper_size[1], forward=True)

    fig.savefig(f"image_scale_1_{scale}.pdf", dpi=dpi)
    plt.close(fig)
```

How to Control the Line Width

The DXF `lineweight` attribute defines the line width as absolute width on the output medium (e.g. 25 = 0.25mm) and therefore depends only on the DPI (dots per inch) setting of the `Figure` class and the `savefig()` method.

There are two additional settings in the *`Configuration`* class which influences the line width:

- *`min_lineweight`* sets the minimum line width in 1/300 inch - a value of 300 is a line width of 1 inch
- *`lineweight_scaling`*, multiply the line width by a this factor

The following table shows the line width in pixels for all valid DXF lineweights for a resolution of 72, 100, 200 and 300 dpi:

		Line Width in Pixels			
Output Resolution [dpi]		**72**	**100**	**200**	**300**
Lineweight					
1/100 mm	**mm**				
5	**0,05**	0,1	0,2	0,4	0,6
9	**0,09**	0,3	0,4	0,7	1,1
13	**0,13**	0,4	0,5	1,0	1,5
15	**0,15**	0,4	0,6	1,2	1,8
18	**0,18**	0,5	0,7	1,4	2,1
20	**0,20**	0,6	0,8	1,6	2,4
25	**0,25**	0,7	1,0	2,0	3,0
30	**0,30**	0,9	1,2	2,4	3,5
35	**0,35**	1,0	1,4	2,8	4,1
40	**0,40**	1,1	1,6	3,1	4,7
50	**0,50**	1,4	2,0	3,9	5,9
53	**0,53**	1,5	2,1	4,2	6,3
60	**0,60**	1,7	2,4	4,7	7,1
70	**0,70**	2,0	2,8	5,5	8,3
80	**0,80**	2,3	3,1	6,3	9,4
90	**0,90**	2,6	3,5	7,1	10,6
100	**1,00**	2,8	3,9	7,9	11,8
106	**1,06**	3,0	4,2	8,3	12,5
120	**1,20**	3,4	4,7	9,4	14,2
140	**1,40**	4,0	5,5	11,0	16,5
158	**1,58**	4,5	6,2	12,4	18,7
200	**2,00**	5,7	7,9	15,7	23,6
211	**2,11**	6,0	8,3	16,6	24,9

See also:

Discussion: https://github.com/mozman/ezdxf/discussions/797

6.8 FAQ

6.8.1 What is the Relationship between ezdxf, dxfwrite and dxfgrabber?

In 2010 I started my first Python package for creating DXF documents called *dxfwrite*, this package can't read DXF files and writes only the DXF R12 (AC1009) version. While *dxfwrite* works fine, I wanted a more versatile package, that can read and write DXF files and maybe also supports newer DXF formats than DXF R12.

This was the start of the *ezdxf* package in 2011, but the progress was so slow, that I created a spin off in 2012 called *dxfgrabber*, which implements only the reading part of *ezdxf*, which I needed for my work and I wasn't sure if *ezdxf* will ever be usable. Luckily in 2014 the first usable version of *ezdxf* could be released. The *ezdxf* package has all the features of *dxfwrite* and *dxfgrabber* and much more, but with a different API. So *ezdxf* is not a drop-in replacement for *dxfgrabber* or *dxfwrite*.

Since *ezdxf* can do all the things that *dxfwrite* and *dxfgrabber* can do, I focused on the development of *ezdxf*, *dxfwrite* and *dxfgrabber* are in maintenance-only mode and will not get any new features, just bugfixes.

There are no advantages of *dxfwrite* over *ezdxf*, *dxfwrite* has a smaller memory footprint, but the `r12writer` add-on does the same job as *dxfwrite* without any in-memory structures by writing direct to a stream or file and there is also no advantage of *dxfgrabber* over *ezdxf* for ordinary DXF files, the smaller memory footprint of *dxfgrabber* is not noticeable and for really big files the `iterdxf` add-on does a better job.

6.8.2 Imported ezdxf package has no content. (readfile, new)

1. AttributeError: partially initialized module 'ezdxf' has no attribute 'readfile' (most likely due to a circular import)

   Did you name your file/script "ezdxf.py"? This causes problems with circular imports. Renaming your file/script should solve this issue.

2. AttributeError: module 'ezdxf' has no attribute 'readfile'

   This could be a hidden permission error, for more information about this issue read Petr Zemeks article: https://blog.petrzemek.net/2020/11/17/when-you-import-a-python-package-and-it-is-empty/

6.8.3 How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?

The BODY, 3DSOLID, SURFACE, REGION and so on, are stored as ACIS data embedded in the DXF file. The ACIS data is stored as SAT (text) format in the entity itself for DXF R2000-R2010 and as SAB (binary) format in the ACDSDATA section for DXF R2013+. *Ezdxf* can read SAT and SAB data, but only write SAT data.

The ACIS data is a proprietary format from Spatial Inc., and there exist no free available documentation or open source libraries to create or edit SAT or SAB data, and also *ezdxf* provides no functionality for creating or editing ACIS data.

The ACIS support provided by *ezdxf* is only useful for users which have access to the ACIS SDK from Spatial Inc..

6.8.4 Are OLE/OLE2 entities supported?

TLDR; NO!

The Wikipedia definition of OLE: Object Linking & Embedding (OLE) is a proprietary technology developed by Microsoft that allows embedding and linking to documents and other objects. For developers, it brought OLE Control Extension (OCX), a way to develop and use custom user interface elements. On a technical level, an OLE object is any object that implements the `IOleObject` interface, possibly along with a wide range of other interfaces, depending on the object's needs.

Therefore *ezdxf* does not support this entities in any way, this only work on Windows and with the required editing application installed. The binary data stored in the OLE objects cannot be used without the editing application.

In my opinion, using OLE objects in a CAD drawing is a very bad design decision that can and will cause problems opening these files in the future, even in AutoCAD on Windows when the required editing application is no longer available or the underlying technology is no longer supported.

All of this is unacceptable for a data storage format that should be accessed for many years or decades (e.g. construction drawings for buildings or bridges).

6.8.5 Rendering SHX fonts

The SHX font format is not documented nor supported by many libraries/packages like *Matplotlib* and *Qt*, therefore only SHX fonts which have corresponding TTF-fonts can be rendered by these backends. See also how-tos about *Fonts*

6.8.6 Drawing Add-on

There is a dedicated how-to section for the *Drawing Add-on*.

6.8.7 Is the AutoCAD command XYZ available?

TLDR; Would you expect Photoshop features from a JPG library?

The package is designed as an interface to the DXF format and therefore does not offer any advanced features of interactive CAD applications. First, some tasks are difficult to perform without human guidance, and second, in complex situations, it's not that easy to tell a "headless" system what exactly to do, so it's very likely that not many users would ever use these features, despite the fact that a lot of time and effort would have to be spent on development, testing and long-term support.

6.9 Reference

The DXF Reference is online available at Autodesk.

Quoted from the original DXF 12 Reference which is not available on the web:

> Since the AutoCAD drawing database (.dwg file) is written in a compact format that changes significantly as new features are added to AutoCAD, we do not document its format and do not recommend that you attempt to write programs to read it directly. To assist in interchanging drawings between AutoCAD and other programs, a Drawing Interchange file format (DXF) has been defined. All implementations of AutoCAD accept this format and are able to convert it to and from their internal drawing file representation.

6.9.1 DXF Document

Document Management

Create New Drawings

ezdxf.**new**(*dxfversion='AC1027'*, *setup=False*, *units=6*) → *Drawing*

Create a new `Drawing` from scratch, *dxfversion* can be either "AC1009" the official DXF version name or "R12" the AutoCAD release name.

new() can create drawings for following DXF versions:

Version	AutoCAD Release
AC1009	AutoCAD R12
AC1015	AutoCAD R2000
AC1018	AutoCAD R2004
AC1021	AutoCAD R2007
AC1024	AutoCAD R2010
AC1027	AutoCAD R2013
AC1032	AutoCAD R2018

The *units* argument defines th document and modelspace units. The header variable $MEASUREMENT will be set according to the given *units*, 0 for inch, feet, miles, … and 1 for metric units. For more information go to module *ezdxf.units*

Parameters

- **dxfversion** – DXF version specifier as string, default is "AC1027" respectively "R2013"
- **setup** – setup default styles, `False` for no setup, `True` to setup everything or a list of topics as strings, e.g. ["linetypes", "styles"] to setup only some topics:

Topic	Description
linetypes	setup line types
styles	setup text styles
dimstyles	setup default *ezdxf* dimension styles
visualstyles	setup 25 standard visual styles

- **units** – document and modelspace units, default is 6 for meters

Open Drawings

Open DXF drawings from file system or text stream, byte stream usage is not supported.

DXF files prior to R2007 requires file encoding defined by header variable $DWGCODEPAGE, DXF R2007 and later requires an UTF-8 encoding.

ezdxf supports reading of files for following DXF versions:

Version	Release	Encoding	Remarks
< AC1009		$DWGCODEPAGE	pre AutoCAD R12 upgraded to AC1009
AC1009	R12	$DWGCODEPAGE	AutoCAD R12
AC1012	R13	$DWGCODEPAGE	AutoCAD R13 upgraded to AC1015
AC1014	R14	$DWGCODEPAGE	AutoCAD R14 upgraded to AC1015
AC1015	R2000	$DWGCODEPAGE	AutoCAD R2000
AC1018	R2004	$DWGCODEPAGE	AutoCAD R2004
AC1021	R2007	UTF-8	AutoCAD R2007
AC1024	R2010	UTF-8	AutoCAD R2010
AC1027	R2013	UTF-8	AutoCAD R2013
AC1032	R2018	UTF-8	AutoCAD R2018

ezdxf.**readfile**(*filename: str | Path*, *encoding: str | None = None*, *errors: str = 'surrogateescape'*) → *Drawing*

Read the DXF document *filename* from the file-system.

This is the preferred method to load existing ASCII or Binary DXF files, the required text encoding will be detected automatically and decoding errors will be ignored.

Override encoding detection by setting argument *encoding* to the estimated encoding. (use Python encoding names like in the `open()` function).

If this function struggles to load the DXF document and raises a `DXFStructureError` exception, try the *ezdxf.recover.readfile()* function to load this corrupt DXF document.

Parameters

- **filename** – filename of the ASCII- or Binary DXF document
- **encoding** – use `None` for auto detect (default), or set a specific encoding like "utf-8", argument is ignored for Binary DXF files
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an `UnicodeDecodeError` exception for invalid data

Raises

- **IOError** – not a DXF file or file does not exist
- ***DXFStructureError*** – for invalid or corrupted DXF structures
- **UnicodeDecodeError** – if *errors* is "strict" and a decoding error occurs

ezdxf.**read**(*stream: TextIO*) → *Drawing*

Read a DXF document from a text-stream. Open stream in text mode (`mode='rt'`) and set correct text encoding, the stream requires at least a `readline()` method.

Since DXF version R2007 (AC1021) file encoding is always "utf-8", use the helper function `dxf_stream_info()` to detect the required text encoding for prior DXF versions. To preserve possible binary data in use `errors='surrogateescape'` as error handler for the import stream.

If this function struggles to load the DXF document and raises a `DXFStructureError` exception, try the *ezdxf.recover.read()* function to load this corrupt DXF document.

Parameters

stream – input text stream opened with correct encoding

Raises

DXFStructureError – for invalid or corrupted DXF structures

`ezdxf.`**`readzip`**(*zipfile: str*, *filename: str | None = None*, *errors: str = 'surrogateescape'*) → *Drawing*

Load a DXF document specified by *filename* from a zip archive, or if *filename* is `None` the first DXF document in the zip archive.

Parameters

- **`zipfile`** – name of the zip archive
- **`filename`** – filename of DXF file, or `None` to load the first DXF document from the zip archive.
- **`errors`** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an `UnicodeDecodeError` exception for invalid data

Raises

- **`IOError`** – not a DXF file or file does not exist or if *filename* is `None` - no DXF file found
- ***DXFStructureError*** – for invalid or corrupted DXF structures
- **`UnicodeDecodeError`** – if *errors* is "strict" and a decoding error occurs

`ezdxf.`**`decode_base64`**(*data: bytes*, *errors: str = 'surrogateescape'*) → *Drawing*

Load a DXF document from base64 encoded binary data, like uploaded data to web applications.

Parameters

- **`data`** – DXF document base64 encoded binary data
- **`errors`** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an `UnicodeDecodeError` exception for invalid data

Raises

- ***DXFStructureError*** – for invalid or corrupted DXF structures
- **`UnicodeDecodeError`** – if *errors* is "strict" and a decoding error occurs

---

Hint: This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the *ezdxf.recover* module.

Save Drawings

Save the DXF document to the file system by *Drawing* methods *save()* or *saveas()*. Write the DXF document to a text stream with *write()*, the text stream requires at least a `write()` method. Get required output encoding for text streams by property `Drawing.output_encoding`

Drawing Settings

The *HeaderSection* stores meta data like modelspace extensions, user name or saving time and current application settings, like actual layer, text style or dimension style settings. These settings are not necessary to process DXF data and therefore many of this settings are not maintained by *ezdxf* automatically.

Header variables set at new

$ACADVER	DXF version
$TDCREATE	date/time at creating the drawing
$FINGERPRINTGUID	every drawing gets a GUID

Header variables updated at saving

$TDUPDATE	actual date/time at saving
$HANDSEED	next available handle as hex string
$DWGCODEPAGE	encoding setting
$VERSIONGUID	every saved version gets a new GUID

See also:

- Howto: *Set/Get Header Variables*
- Howto: *Set DXF Drawing Units*

Ezdxf Metadata

Store internal metadata like *ezdxf* version and creation time for a new created document as metadata in the DXF file. Only standard DXF features are used to store meta data and this meta data is preserved by Autodesk products, BricsCAD and of course *ezdxf*. Other 3rd party DXF libraries may remove this meta data.

For DXF R12 the meta data is stored as XDATA by AppID `EZDXF` in the model space BLOCK entity in the BLOCKS section.

For DXF R2000+ the meta data is stored in the "root" DICTIONARY in the OBJECTS section as a DICTIONARY object by the key `EZDXF_META`.

The `MetaData` object has a dict-like interface and can also store custom metadata:

```
metadata = doc.ezdxf_metadata()

# set data
metadata["MY_CUSTOM_META_DATA"] = "a string with max. length of 254"
```

(continues on next page)

(continued from previous page)

```
# get data, raises a KeyError() if key not exist
value = metadata["MY_CUSTOM_META_DATA"]

# get data, returns an empty string if key not exist
value = metadata.get("MY_CUSTOM_META_DATA")

# delete entry, raises a KeyError() if key not exist
del metadata["MY_CUSTOM_META_DATA"]

# discard entry, does not raise a KeyError() if key not exist
metadata.discard("MY_CUSTOM_META_DATA")
```

Keys and values are limited to strings with a max. length of 254 characters and line ending `\n` will be replaced by `\P`.

Keys used by *ezdxf*:

- `WRITTEN_BY_EZDXF`: *ezdxf* version and UTC time in ISO format
- `CREATED_BY_EZDXF`: *ezdxf* version and UTC time in ISO format

Example of the ezdxf marker string: `0.16.4b1 @ 2021-06-12T07:35:34.898808+00:00`

class ezdxf.document.**MetaData**

abstract MetaData.**__contains__**(*key: str*) → bool

Returns *key* `in` self.

abstract MetaData.**__getitem__**(*key: str*) → str

Returns the value for self[*key*].

Raises

KeyError – *key* does not exist

MetaData.**get**(*key: str*, *default: str = ''*) → str

Returns the value for *key*. Returns *default* if *key* not exist.

abstract MetaData.**__setitem__**(*key: str*, *value: str*) → None

Set self[*key*] to *value*.

abstract MetaData.**__delitem__**(*key: str*) → None

Delete self[*key*].

Raises

KeyError – *key* does not exist

MetaData.**discard**(*key: str*) → None

Remove *key*, does **not** raise an exception if *key* not exist.

Drawing Class

The *Drawing* class is the central management structure of a DXF document.

Access Layouts

- *Drawing.modelspace()*
- *Drawing.paperspace()*

Access Resources

- Application ID Table: *Drawing.appids*
- Block Definition Table: *Drawing.blocks*
- Dimension Style Table: *Drawing.dimstyles*
- Layer Table: *Drawing.layers*
- Linetype Table: *Drawing.linetypes*
- MLeader Style Table: *Drawing.mleader_styles*
- MLine Style Table: *Drawing.mline_styles*
- Material Table: *Drawing.materials*
- Text Style Table: *Drawing.styles*
- UCS Table: *Drawing.ucs*
- VPort Table: *Drawing.viewports*
- View Table: *Drawing.views*
- Classes Section: *Drawing.classes*
- Object Section: *Drawing.objects*
- Entity Database: `Drawing.entitydb`
- Entity Groups: *Drawing.groups*
- Header Variables: *Drawing.header*

Drawing Class

class ezdxf.document.**Drawing**

The *Drawing* class is the central management structure of a DXF document.

dxfversion

Actual DXF version like `'AC1009'`, set by *ezdxf.new()* or *ezdxf.readfile()*.

For supported DXF versions see *Document Management*

acad_release

The AutoCAD release name like `'R12'` or `'R2000'` for actual *dxfversion*.

encoding

Text encoding of *Drawing*, the default encoding for new drawings is `'cp1252'`. Starting with DXF R2007 (AC1021), DXF files are written as UTF-8 encoded text files, regardless of the attribute *encoding*. The text encoding can be changed to encodings listed below.

see also: *DXF File Encoding*

supported	encodings
`'cp874'`	Thai
`'cp932'`	Japanese
`'gbk'`	UnifiedChinese
`'cp949'`	Korean
`'cp950'`	TradChinese
`'cp1250'`	CentralEurope
`'cp1251'`	Cyrillic
`'cp1252'`	WesternEurope
`'cp1253'`	Greek
`'cp1254'`	Turkish
`'cp1255'`	Hebrew
`'cp1256'`	Arabic
`'cp1257'`	Baltic
`'cp1258'`	Vietnam

output_encoding

Returns required output encoding for saving to filesystem or encoding to binary data.

filename

Drawing filename, if loaded by *ezdxf.readfile()* else `None`.

rootdict

Reference to the root dictionary of the OBJECTS section.

header

Reference to the *HeaderSection*, get/set drawing settings as header variables.

entities

Reference to the `EntitySection` of the drawing, where all graphical entities are stored, but only from modelspace and the *active* paperspace layout. Just for your information: Entities of other paperspace layouts are stored as *BlockLayout* in the *BlocksSection*.

objects

Reference to the objects section, see also *ObjectsSection*.

blocks

Reference to the blocks section, see also *BlocksSection*.

tables

Reference to the tables section, see also *TablesSection*.

classes

Reference to the classes section, see also *ClassesSection*.

layouts

Reference to the layout manager, see also *Layouts*.

groups

Collection of all groups, see also *GroupCollection*.

requires DXF R13 or later

layers

Shortcut for `Drawing.tables.layers`

Reference to the layers table, where you can create, get and remove layers, see also *Table* and *Layer*

styles

Shortcut for `Drawing.tables.styles`

Reference to the styles table, see also *Textstyle*.

dimstyles

Shortcut for `Drawing.tables.dimstyles`

Reference to the dimstyles table, see also *DimStyle*.

linetypes

Shortcut for `Drawing.tables.linetypes`

Reference to the linetypes table, see also *Linetype*.

views

Shortcut for `Drawing.tables.views`

Reference to the views table, see also *View*.

viewports

Shortcut for `Drawing.tables.viewports`

Reference to the viewports table, see also *VPort*.

ucs

Shortcut for `Drawing.tables.ucs`

Reference to the ucs table, see also *UCSTableEntry*.

appids

Shortcut for `Drawing.tables.appids`

Reference to the appids table, see also *AppID*.

materials

`MaterialCollection` of all `Material` objects.

mline_styles

`MLineStyleCollection` of all *MLineStyle* objects.

mleader_styles

`MLeaderStyleCollection` of all *MLeaderStyle* objects.

units

Get and set the document/modelspace base units as enum, for more information read this: *DXF Units*.

get_abs_filepath = <function Drawing.get_abs_filepath>

save (*encoding: str | None = None*, *fmt: str = 'asc'*) → None

Write drawing to file-system by using the *filename* attribute as filename. Override file encoding by argument *encoding*, handle with care, but this option allows you to create DXF files for applications that handle file encoding different from AutoCAD.

Parameters

- **encoding** – override default encoding as Python encoding string like `'utf-8'`
- **fmt** – `'asc'` for ASCII DXF (default) or `'bin'` for Binary DXF

saveas (*filename: PathLike | str*, *encoding: str | None = None*, *fmt: str = 'asc'*) → None

Set *Drawing* attribute *filename* to *filename* and write drawing to the file system. Override file encoding by argument *encoding*, handle with care, but this option allows you to create DXF files for applications that handles file encoding different than AutoCAD.

Parameters

- **filename** – file name as string
- **encoding** – override default encoding as Python encoding string like `'utf-8'`
- **fmt** – `'asc'` for ASCII DXF (default) or `'bin'` for Binary DXF

write (*stream: TextIO | BinaryIO*, *fmt: str = 'asc'*) → None

Write drawing as ASCII DXF to a text stream or as Binary DXF to a binary stream. For DXF R2004 (AC1018) and prior open stream with drawing *encoding* and `mode='wt'`. For DXF R2007 (AC1021) and later use `encoding='utf-8'`, or better use the later added *Drawing* property *output_encoding* which returns the correct encoding automatically. The correct and required error handler is `errors='dxfreplace'`!

If writing to a `StringIO` stream, use *Drawing.encode()* to encode the result string from `StringIO.get_value()`:

```
binary = doc.encode(stream.get_value())
```

Parameters

- **stream** – output text stream or binary stream
- **fmt** – "asc" for ASCII DXF (default) or "bin" for binary DXF

encode_base64 () → bytes

Returns DXF document as base64 encoded binary data.

encode (*s: str*) → bytes

Encode string *s* with correct encoding and error handler.

query (*query: str = '*'*) → *EntityQuery*

Entity query over all layouts and blocks, excluding the OBJECTS section and the resource tables of the TABLES section.

Parameters
query – query string

See also:

Entity Query String and *Retrieve entities by query language*

groupby(*dxfattrib=''*, *key=None*) → dict

Groups DXF entities of all layouts and blocks (excluding the OBJECTS section) by a DXF attribute or a key function.

Parameters

- **dxfattrib** – grouping DXF attribute like "layer"
- **key** – key function, which accepts a `DXFEntity` as argument and returns a hashable grouping key or `None` to ignore this entity.

See also:

groupby() documentation

modelspace() → Modelspace

Returns the modelspace layout, displayed as "Model" tab in CAD applications, defined by block record named "*Model_Space".

paperspace(*name: str = ''*) → Paperspace

Returns paperspace layout *name* or the active paperspace if no name is given.

Parameters
name – paperspace name or empty string for the active paperspace

Raises
KeyError – if the modelspace was acquired or layout *name* does not exist

layout(*name: str = ''*) → *Layout*

Returns paperspace layout *name* or the first layout in tab-order if no name is given.

Parameters
name – paperspace name or empty string for the first paperspace in tab-order

Raises
KeyError – layout *name* does not exist

active_layout() → Paperspace

Returns the active paperspace layout, defined by block record name "*Paper_Space".

layout_names() → Iterable[str]

Returns all layout names in arbitrary order.

layout_names_in_taborder() → Iterable[str]

Returns all layout names in tab-order, "Model" is always the first name.

new_layout(*name*, *dxfattribs=None*) → Paperspace

Create a new paperspace layout *name*. Returns a *Paperspace* object. DXF R12 (AC1009) supports only one paperspace layout, only the active paperspace layout is saved, other layouts are dismissed.

Parameters

- **name** – unique layout name
- **dxfattribs** – additional DXF attributes for the `DXFLayout` entity

Raises
DXFValueError – paperspace layout *name* already exist

page_setup(*name: str = 'Layout1'*, *fmt: str = 'ISO A3'*, *landscape=True*) → Paperspace

Creates a new paperspace layout if *name* does not exist or reset the existing layout. This method requires DXF R2000 or newer. The paper format name *fmt* defines one of the following paper sizes, measures in landscape orientation:

Name	Units	Width	Height
ISO A0	mm	1189	841
ISO A1	mm	841	594
ISO A2	mm	594	420
ISO A3	mm	420	297
ISO A4	mm	297	210
ANSI A	inch	11	8.5
ANSI B	inch	17	11
ANSI C	inch	22	17
ANSI D	inch	34	22
ANSI E	inch	44	34
ARCH C	inch	24	18
ARCH D	inch	36	24
ARCH E	inch	48	36
ARCH E1	inch	42	30
Letter	inch	11	8.5
Legal	inch	14	8.5

The layout uses the associated units of the paper format as drawing units, has no margins or offset defined and the scale of the paperspace layout is 1:1.

Parameters

- **name** – paperspace layout name
- **fmt** – paper format
- **landscape** – `True` for landscape orientation, `False` for portrait orientation

delete_layout(*name: str*) → None

Delete paper space layout *name* and all entities owned by this layout. Available only for DXF R2000 or later, DXF R12 supports only one paperspace, and it can't be deleted.

add_image_def(*filename: str*, *size_in_pixel: tuple[int, int]*, *name=None*)

Add an image definition to the objects section.

Add an `ImageDef` entity to the drawing (objects section). *filename* is the image file name as relative or absolute path and *size_in_pixel* is the image size in pixel as (x, y) tuple. To avoid dependencies to external packages, *ezdxf* can not determine the image size by itself. Returns a `ImageDef` entity which is needed to create an image reference. *name* is the internal image name, if set to `None`, name is auto-generated.

Absolute image paths works best for AutoCAD but not perfect, you have to update external references manually in AutoCAD, which is not possible in TrueView. If the drawing units differ from 1 meter, you also have to use: *set_raster_variables()*.

Parameters

- **filename** – image file name (absolute path works best for AutoCAD)
- **size_in_pixel** – image size in pixel as (x, y) tuple
- **name** – image name for internal use, None for using filename as name (best for AutoCAD)

See also:

Tutorial for Image and ImageDef

set_raster_variables (*frame: int = 0*, *quality: int = 1*, *units: str = 'm'*)

Set raster variables.

Parameters

- **frame** – 0 = do not show image frame; 1 = show image frame
- **quality** – 0 = draft; 1 = high
- **units** – units for inserting images. This defines the real world unit for one drawing unit for the purpose of inserting and scaling images with an associated resolution.

mm	Millimeter
cm	Centimeter
m	Meter (ezdxf default)
km	Kilometer
in	Inch
ft	Foot
yd	Yard
mi	Mile

set_wipeout_variables (*frame=0*)

Set wipeout variables.

Parameters

frame – 0 = do not show image frame; 1 = show image frame

add_underlay_def (*filename: str*, *fmt: str = 'ext'*, *name: str | None = None*)

Add an `UnderlayDef` entity to the drawing (OBJECTS section). The *filename* is the underlay file name as relative or absolute path and *fmt* as string (pdf, dwf, dgn). The underlay definition is required to create an underlay reference.

Parameters

- **filename** – underlay file name
- **fmt** – file format as string "pdf"|"dwf"|"dgn" or "ext" for getting file format from filename extension
- **name** – pdf format = page number to display; dgn format = "default"; dwf: ????

See also:

Tutorial for Underlay and UnderlayDefinition

add_xref_def (*filename: str*, *name: str*, *flags: int = BLK_XREF | BLK_EXTERNAL*)

Add an external reference (xref) definition to the blocks section.

Parameters

- **filename** – external reference filename
- **name** – name of the xref block
- **flags** – block flags

layouts_and_blocks() → Iterator[GenericLayoutType]

Iterate over all layouts (modelspace and paperspace) and all block definitions.

chain_layouts_and_blocks() → Iterator[*DXFEntity*]

Chain entity spaces of all layouts and blocks. Yields an iterator for all entities in all layouts and blocks.

reset_fingerprint_guid()

Reset fingerprint GUID.

reset_version_guid()

Reset version GUID.

set_modelspace_vport(*height*, *center=(0, 0)*, *, *dxfattribs=None*) → *VPort*

Set initial view/zoom location for the modelspace, this replaces the current "*Active" viewport configuration (`VPort`) and reset the coordinate system to the *WCS*.

Parameters

- **height** – modelspace area to view
- **center** – modelspace location to view in the center of the CAD application window.
- **dxfattribs** – additional DXF attributes for the VPORT entity

audit() → Auditor

Checks document integrity and fixes all fixable problems, not fixable problems are stored in `Auditor.errors`.

If you are messing around with internal structures, call this method before saving to be sure to export valid DXF documents, but be aware this is a long-running task.

validate(*print_report=True*) → bool

Simple way to run an audit process. Fixes all fixable problems, return `False` if not fixable errors occurs. Prints a report of resolved and unrecoverable errors, if requested.

Parameters

print_report – print report to stdout

Returns: `False` if unrecoverable errors exist

ezdxf_metadata() → *MetaData*

Returns the *ezdxf* `ezdxf.document.MetaData` object, which manages *ezdxf* and custom metadata in DXF files. For more information see: *Ezdxf Metadata*.

Recover

This module provides functions to "recover" ASCII DXF documents with structural flaws, which prevents the regular `ezdxf.read()` and `ezdxf.readfile()` functions to load the document.

The `read()` and `readfile()` functions will repair as much flaws as possible and run the required audit process automatically afterwards and return the result of this audit process:

```
import sys
import ezdxf
from ezdxf import recover

try:
    doc, auditor = recover.readfile("messy.dxf")
except IOError:
```

(continues on next page)

(continued from previous page)

```
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe just
# a problem when saving the recovered DXF file.
if auditor.has_errors:
    auditor.print_error_report()
```

The loading functions also decode DXF-Unicode encoding automatically e.g. "\U+00FC" -> "ü". All these efforts cost some time, loading the DXF document with *ezdxf.read()* or *ezdxf.readfile()* is faster.

Warning: This module will load DXF files which have decoding errors, most likely binary data stored in XRECORD entities, these errors are logged as unrecoverable `AuditError.DECODE_ERRORS` in the `Auditor.errors` attribute, but no `DXFStructureError` exception will be raised, because for many use cases this errors can be ignored.

Writing such files back with *ezdxf* may create **invalid** DXF files, or at least some **information will be lost** - handle with care!

To avoid this problem use `recover.readfile(filename, errors='strict')` which raises an `UnicodeDecodeError` exception for such binary data. Catch the exception and handle this DXF files as unrecoverable.

Loading Scenarios

1. It will work

Mostly DXF files from AutoCAD or BricsCAD (e.g. for In-house solutions):

```
try:
    doc = ezdxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
```

2. DXF file with minor flaws

DXF files have only minor flaws, like undefined resources:

```
try:
    doc = ezdxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
```

(continues on next page)

(continued from previous page)

```
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)

auditor = doc.audit()
if auditor.has_errors:
    auditor.print_error_report()
```

3. Try Hard

From trusted and untrusted sources but with good hopes, the worst case works like a cache miss, you pay for the first try and pay the extra fee for the recover mode:

```
try:  # Fast path:
    doc = ezdxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
# Catch all DXF errors:
except ezdxf.DXFError:
    try:  # Slow path including fixing low level structures:
        doc, auditor = recover.readfile(name)
    except ezdxf.DXFStructureError:
        print(f'Invalid or corrupted DXF file: {name}.')
        sys.exit(2)

    # DXF file can still have unrecoverable errors, but this is maybe
    # just a problem when saving the recovered DXF file.
    if auditor.has_errors:
        print(f'Found unrecoverable errors in DXF file: {name}.')
        auditor.print_error_report()
```

4. Just use the slow recover module

Untrusted sources and expecting many invalid or corrupted DXF files, you always pay an extra fee for the recover mode:

```
try:  # Slow path including fixing low level structures:
    doc, auditor = recover.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.')
    auditor.print_error_report()
```

5. Unrecoverable Decoding Errors

If files contain binary data which can not be decoded by the document encoding, it is maybe the best to ignore these files, this works in normal and recover mode:

```
try:
    doc, auditor = recover.readfile(name, errors='strict')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
except UnicodeDecodeError:
    print(f'Decoding error in DXF file: {name}.')
    sys.exit(3)
```

6. Ignore/Locate Decoding Errors

Sometimes ignoring decoding errors can recover DXF files or at least you can detect where the decoding errors occur:

```
try:
    doc, auditor = recover.readfile(name, errors='ignore')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
if auditor.has_errors:
    auditor.print_report()
```

The error messages with code `AuditError.DECODING_ERROR` shows the approximate line number of the decoding error: "Fixed unicode decoding error near line: xxx."

Hint: This functions can handle only ASCII DXF files!

`ezdxf.recover.`**`readfile`**(*filename: str* | Path, *errors: str* = *'surrogateescape'*) → tuple[*Drawing*, Auditor]

Read a DXF document from file system similar to *`ezdxf.readfile()`*, but this function will repair as many flaws as possible, runs the required audit process automatically the DXF document and the `Auditor`.

Parameters

- **`filename`** – file-system name of the DXF document to load
- **`errors`** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an `UnicodeDecodeError` exception for invalid data

Raises

- ***`DXFStructureError`*** – for invalid or corrupted DXF structures

- **UnicodeDecodeError** – if *errors* is "strict" and a decoding error occurs

ezdxf.recover.**read**(*stream: BinaryIO*, *errors: str = 'surrogateescape'*) → tuple[*Drawing*, Auditor]

Read a DXF document from a binary-stream similar to *ezdxf.read()*, but this function will detect the text encoding automatically and repair as many flaws as possible, runs the required audit process afterwards and returns the DXF document and the Auditor.

Parameters

- **stream** – data stream to load in binary read mode
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an UnicodeDecodeError exception for invalid data

Raises

- ***DXFStructureError*** – for invalid or corrupted DXF structures
- **UnicodeDecodeError** – if *errors* is "strict" and a decoding error occurs

ezdxf.recover.**explore**(*filename: str | Path*, *errors: str = 'ignore'*) → tuple[*Drawing*, Auditor]

Read a DXF document from file system similar to *readfile()*, but this function will use a special tag loader, which tries to recover the tag stream if invalid tags occur. This function is intended to load corrupted DXF files and should only be used to explore such files, data loss is very likely.

Parameters

- **filename** – file-system name of the DXF document to load
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an UnicodeDecodeError exception for invalid data

Raises

- ***DXFStructureError*** – for invalid or corrupted DXF structures
- **UnicodeDecodeError** – if *errors* is "strict" and a decoding error occurs

r12strict

New in version 1.1.

Due to ACAD release 14 the resource names, such as layer-, linetype, text style-, dimstyle- and block names, were limited to 31 characters in length and all names were uppercase.

Names can include the letters A to Z, the numerals 0 to 9, and the special characters, dollar sign "$", underscore "_", hyphen "-" and the asterix "*" as first character for special names like anonymous blocks. Most applications do not care about that and work fine with longer names and any characters used in names for some exceptions, but of course Autodesk applications are very picky about that.

The function *make_acad_compatible()* makes DXF R12 drawings to 100% compatible to Autodesk products and does everything at once, but the different processing steps can be called manually.

Important: This module can only process DXF R12 file and will throw a `DXFVersionError` otherwise. For exporting any DXF document as DXF R12 use the *ezdxf.addons.r12export* add-on.

Usage

```
import ezdxf
from ezdxf import r12strict

doc = ezdxf.readfile("r12sloppy.dxf")
r12strict.make_acad_compatible(doc)
doc.saveas("r12strict.dxf")
```

Functions

make_acad_compatible	Apply all DXF R12 requirements, so Autodesk products will load the document.
translate_names	Translate table and block names into strict DXF R12 names.
clean	Removes all features that are not supported for DXF R12 by Autodesk products.

ezdxf.r12strict.**make_acad_compatible**(*doc:* Drawing) → None

Apply all DXF R12 requirements, so Autodesk products will load the document.

ezdxf.r12strict.**translate_names**(*doc:* Drawing) → None

Translate table and block names into strict DXF R12 names.

ACAD Releases upto 14 limit names to 31 characters in length and all names are uppercase. Names can include the letters A to Z, the numerals 0 to 9, and the special characters, dollar sign ($), underscore (_), hyphen (-) and the asterix (*) as first character for special names like anonymous blocks.

Most applications do not care about that and work fine with longer names and any characters used in names for some exceptions, but of course Autodesk applications are very picky about that.

Note: This is a destructive process and modifies the internals of the DXF document.

ezdxf.r12strict.**clean**(*doc:* Drawing) → None

Removes all features that are not supported for DXF R12 by Autodesk products.

class ezdxf.r12strict.**R12NameTranslator**

Translate table and block names into strict DXF R12 names.

ACAD Releases upto 14 limit names to 31 characters in length and all names are uppercase. Names can include the letters A to Z, the numerals 0 to 9, and the special characters, dollar sign ($), underscore (_), hyphen (-) and the asterix (*) as first character for special names like anonymous blocks.

reset() → None

translate(*name: str*) → str

Application Settings

This is a high-level module for working with CAD application settings and behaviors. None of these settings have any influence on the behavior of *ezdxf*, since *ezdxf* only takes care of the content of the DXF file and not of the way it is presented to the user.

Important: You need to understand that these settings work at the application level, *ezdxf* cannot force an application to do something in a certain way! The functionality of this module has been tested with Autodesk TrueView and BricsCAD, other applications may show different results or ignore the settings.

Set Current Properties

The current properties are used by the CAD application to create new entities, these settings do not affect how *ezdxf* creates new entities.

The module `ezdxf.gfxattribs` provides the class `GfxAttribs()`, which can load the current graphical entity settings from the HEADER section for creating new entities by *ezdxf*: `load_from_header()`

`ezdxf.appsettings.`**`set_current_layer`**(*doc:* Drawing, *name: str*)

Set current layer.

`ezdxf.appsettings.`**`set_current_color`**(*doc:* Drawing, *color: int*)

Set current *AutoCAD Color Index (ACI)*.

`ezdxf.appsettings.`**`set_current_linetype`**(*doc:* Drawing, *name: str*)

Set current linetype.

`ezdxf.appsettings.`**`set_current_lineweight`**(*doc:* Drawing, *lineweight: int*)

Set current lineweight, see *Lineweights* reference for valid values.

`ezdxf.appsettings.`**`set_current_linetype_scale`**(*doc:* Drawing, *scale: float*)

Set current linetype scale.

`ezdxf.appsettings.`**`set_current_textstyle`**(*doc:* Drawing, *name: str*)

Set current text style.

`ezdxf.appsettings.`**`set_current_dimstyle`**(*doc:* Drawing, *name: str*)

Set current dimstyle.

`ezdxf.appsettings.`**`set_current_dimstyle_attribs`**(*doc:* Drawing, *name: str*)

Set current dimstyle and copy all dimstyle attributes to the HEADER section.

`ezdxf.appsettings.`**`set_lineweight_display_style`**(*doc:* Drawing, *end_caps:* EndCaps, *join_style:* JoinStyle) → None

Set the style of end caps and joints for linear entities when displaying line weights. These settings only affect objects created afterwards.

Restore the WCS

ezdxf.appsettings.**restore_wcs**(*doc:* Drawing)

Restore the UCS settings in the HEADER section to the *WCS* and reset all active viewports to the WCS.

Update Extents

ezdxf.appsettings.**update_extents**(*doc:* Drawing) → *BoundingBox*

Calculate the extents of the model space, update the HEADER variables $EXTMIN and $EXTMAX and returns the result as *ezdxf.math.BoundingBox*. Note that this function uses the *ezdxf.bbox* module to calculate the extent of the model space. This module is not very fast and not very accurate for text and ignores all *ACIS* based entities.

The function updates only the values in the HEADER section, to zoom the active viewport to this extents, use this recipe:

```
import ezdxf
from ezdxf import zoom, appsettings

doc = ezdxf.readfile("your.dxf")
extents = appsettings.update_extents(doc)
zoom.center(doc.modelspace(), extents.center, extents.size)
```

See also:

- the *ezdxf.bbox* module to understand the limitations of the extent calculation
- the *ezdxf.zoom* module

Show Lineweight

ezdxf.appsettings.**show_lineweight**(*doc:* Drawing, *state=True*) → None

The CAD application or DXF viewer should show lines and curves with "thickness" (lineweight) if *state* is `True`.

6.9.2 DXF Structures

Sections

Header Section

The drawing settings are stored in the HEADER section, which is accessible by the *header* attribute of the *Drawing* object. See the online documentation from Autodesk for available header variables.

See also:

DXF Internals: *HEADER Section*

class ezdxf.sections.header.**HeaderSection**

custom_vars

Stores the custom drawing properties in a *CustomVars* object.

__len__() → int

Returns count of header variables.

__contains__(*key*) → bool

Returns `True` if header variable *key* exist.

varnames() → KeysView

Returns an iterable of all header variable names.

get(*key: str*, *default: Any = None*) → Any

Returns value of header variable *key* if exist, else the *default* value.

__getitem__(*key: str*) → Any

Get header variable *key* by index operator like: `drawing.header['$ACADVER']`

__setitem__(*key: str*, *value: Any*) → None

Set header variable *key* to *value* by index operator like: `drawing.header['$ANGDIR'] = 1`

__delitem__(*key: str*) → None

Delete header variable *key* by index operator like: `del drawing.header['$ANGDIR']`

reset_wcs()

Reset the current UCS settings to the *WCS*.

class ezdxf.sections.header.**CustomVars**

The *CustomVars* class stores custom properties in the DXF header as $CUSTOMPROPERTYTAG and $CUSTOMPROPERTY values. Custom properties require DXF R2004 or later, *ezdxf* can create custom properties for older DXF versions as well, but AutoCAD will not show that properties.

properties

A list of custom header properties, stored as string tuples `(tag, value)`. Multiple occurrence of the same custom tag is allowed, but not well supported by the interface. This is a standard Python list and it's safe to modify this list as long as you just use tuples of strings.

__len__() → int

Count of custom properties.

__iter__() → Iterator[tuple[str, str]]

Iterate over all custom properties as `(tag, value)` tuples.

clear() → None

Remove all custom properties.

get(*tag: str*, *default: str | None = None*)

Returns the value of the first custom property *tag*.

has_tag(*tag: str*) → bool

Returns `True` if custom property *tag* exist.

append(*tag: str*, *value: str*) → None

Add custom property as `(tag, value)` tuple.

replace(*tag: str*, *value: str*) → None

Replaces the value of the first custom property *tag* by a new *value*.

Raises `DXFValueError` if *tag* does not exist.

remove(*tag: str*, *all: bool = False*) → None

Removes the first occurrence of custom property *tag*, removes all occurrences if *all* is `True`.

Raises *:class:`DXFValueError* if *tag* does not exist.

Classes Section

The CLASSES section in DXF files holds the information for application-defined classes whose instances appear in *Layout* objects. As usual package user there is no need to bother about CLASSES.

See also:

DXF Internals: *CLASSES Section*

class ezdxf.sections.classes.**ClassesSection**

classes

Storage of all *DXFClass* objects, they are not stored in the entities database, because CLASS instances do not have a handle attribute.

register()

add_class(*name: str*)

Register a known class by *name*.

get(*name: str*) → DXFClass

Returns the first class matching *name*.

Storage key is the `(name, cpp_class_name)` tuple, because there are some classes with the same `name` but different `cpp_class_names`.

add_required_classes(*dxfversion: str*) → None

Add all required CLASS definitions for the specified DXF version.

update_instance_counters() → None

Update CLASS instance counter for all registered classes, requires DXF R2004+.

class ezdxf.entities.**DXFClass**

Information about application-defined classes.

dxf.**name**

Class DXF record name.

dxf.**cpp_class_name**

C++ class name. Used to bind with software that defines object class behavior.

dxf.**app_name**

Application name. Posted in Alert box when a class definition listed in this section is not currently loaded.

dxf.**flags**

Proxy capabilities flag

0	No operations allowed (0)
1	Erase allowed (0x1)
2	Transform allowed (0x2)
4	Color change allowed (0x4)
8	Layer change allowed (0x8)
16	Linetype change allowed (0x10)
32	Linetype scale change allowed (0x20)
64	Visibility change allowed (0x40)
128	Cloning allowed (0x80)
256	Lineweight change allowed (0x100)
512	Plot Style Name change allowed (0x200)
895	All operations except cloning allowed (0x37F)
1023	All operations allowed (0x3FF)
1024	Disables proxy warning dialog (0x400)
32768	R13 format proxy (0x8000)

`dxf.`**`instance_count`**

Instance count for a custom class.

`dxf.`**`was_a_proxy`**

Set to `1` if class was not loaded when this DXF file was created, and `0` otherwise.

`dxf.`**`is_an_entity`**

Set to `1` if class was derived from the *`DXFGraphic`* class and can reside in layouts. If `0`, instances may appear only in the OBJECTS section.

`key`

Unique name as `(name, cpp_class_name)` tuple.

Tables Section

The TABLES section is the home of all TABLE objects of a DXF document.

See also:

DXF Internals: *TABLES Section*

`class` `ezdxf.sections.tables.`**`TablesSection`**

`layers`

`LayerTable` maintaining the *`Layer`* objects

`linetypes`

`LinetypeTable` maintaining the *`Linetype`* objects

`styles`

`TextstyleTable` maintaining the *`Textstyle`* objects

`dimstyles`

`DimStyleTable` maintaining the *`DimStyle`* objects

`appids`

`AppIDTable` maintaining the *`AppID`* objects

ucs

UCSTable maintaining the UCSTable objects

views

ViewTable maintaining the *View* objects

viewports

ViewportTable maintaining the *VPort* objects

block_records

BlockRecordTable maintaining the *BlockRecord* objects

Blocks Section

The BLOCKS section is the home all block definitions (*BlockLayout*) of a DXF document.

Warning: Blocks are an essential building block of the DXF format. Most blocks are referenced are by name, and renaming or deleting a block is not as easy as it seems, since there is no overall index where all block references appear, and such block references can also reside in custom data or even custom entities, therefore renaming or deleting block definitions can damage a DXF file!

See also:

DXF Internals: *BLOCKS Section* and *Block Management Structures*

class ezdxf.sections.blocks.**BlocksSection**

__iter__() → Iterator[BlockLayout]

Iterable of all *BlockLayout* objects.

__contains__(*name: str*) → bool

Returns True if *BlockLayout* *name* exist.

__getitem__(*name: str*) → BlockLayout

Returns *BlockLayout* *name*, raises DXFKeyError if *name* not exist.

__delitem__(*name: str*) → None

Deletes *BlockLayout* *name* and all of its content, raises DXFKeyError if *name* not exist.

get(*name: str*, *default=None*) → BlockLayout

Returns *BlockLayout* *name*, returns *default* if *name* not exist.

new(*name: str*, *base_point:* UVec = *NULLVEC*, *dxfattribs=None*) → BlockLayout

Create and add a new *BlockLayout*, *name* is the BLOCK name, *base_point* is the insertion point of the BLOCK.

new_anonymous_block(*type_char: str = 'U'*, *base_point:* UVec = *NULLVEC*) → BlockLayout

Create and add a new anonymous *BlockLayout*, *type_char* is the BLOCK type, *base_point* is the insertion point of the BLOCK.

type_char	Anonymous Block Type
'U'	'*U###' anonymous BLOCK
'E'	'*E###' anonymous non-uniformly scaled BLOCK
'X'	'*X###' anonymous HATCH graphic
'D'	'*D###' anonymous DIMENSION graphic
'A'	'*A###' anonymous GROUP
'T'	'*T###' anonymous block for ACAD_TABLE content

rename_block(*old_name: str*, *new_name: str*) → None

Rename *BlockLayout* *old_name* to *new_name*

> **Warning:** This is a low-level tool and does not rename the block references, so all block references to *old_name* are pointing to a non-existing block definition!

delete_block(*name: str*, *safe: bool = True*) → None

Delete block. Checks if the block is still referenced if *safe* is `True`.

Parameters

- **name** – block name (case insensitive)
- **safe** – check if the block is still referenced or a special block without explicit references

Raises

- ***DXFKeyError*** – if block not exists
- ***DXFBlockInUseError*** – if block is still referenced, and safe is `True`

delete_all_blocks() → None

Delete all blocks without references except modelspace- or paperspace layout blocks, special arrow- and anonymous blocks (DIMENSION, ACAD_TABLE).

> **Warning:** There could exist references to blocks which are not documented in the DXF reference, hidden in extended data sections or application defined data, which could invalidate a DXF document if these blocks will be deleted.

Entities Section

The ENTITIES section is the home of all entities of the *Modelspace* and the active *Paperspace* layout. This is a real section in the DXF file but in *ezdxf* the *EntitySection* is just a linked entity space of these two layouts.

See also:

DXF Internals: *ENTITIES Section*

class ezdxf.sections.entities.**EntitySection**

__iter__() → Iterator[*DXFEntity*]

Returns an iterator for all entities of the modelspace and the active paperspace.

__len__() → int

Returns the count of all entities in the modelspace and the active paperspace.

Objects Section

The OBJECTS section is the home of all none graphical objects of a DXF document. The OBJECTS section is accessible by the `Drawing.objects` attribute.

Convenience methods of the *Drawing* object to create essential structures in the OBJECTS section:

- IMAGEDEF: *add_image_def()*
- UNDERLAYDEF: *add_underlay_def()*
- RASTERVARIABLES: *set_raster_variables()*
- WIPEOUTVARIABLES: *set_wipeout_variables()*

See also:

DXF Internals: *OBJECTS Section*

class ezdxf.sections.objects.**ObjectsSection**

rootdict

Returns the root DICTIONARY, or as AutoCAD calls it: the named DICTIONARY.

__len__() → int

Returns the count of all DXF objects in the OBJECTS section.

__iter__() → Iterator[*DXFObject*]

Returns an iterator of all DXF objects in the OBJECTS section.

__getitem__(*index*) → *DXFObject*

Get entity at *index*.

The underlying data structure for storing DXF objects is organized like a standard Python list, therefore *index* can be any valid list indexing or slicing term, like a single index `objects[-1]` to get the last entity, or an index slice `objects[:10]` to get the first 10 or fewer objects as `list[DXFObject]`.

__contains__(*entity*)

Returns `True` if *entity* stored in OBJECTS section.

Parameters
entity – `DXFObject` or handle as hex string

query(*query: str = '*'*) → *EntityQuery*

Get all DXF objects matching the *Entity Query String*.

add_dictionary(*owner: str = '0'*, *hard_owned: bool = True*) → Dictionary

Add new *Dictionary* object.

Parameters

- **owner** – handle to owner as hex string.
- **hard_owned** – `True` to treat entries as hard owned.

add_dictionary_with_default(*owner='0'*, *default='0'*, *hard_owned: bool = True*) → *DictionaryWithDefault*

Add new *DictionaryWithDefault* object.

Parameters

- **owner** – handle to owner as hex string.

- **default** – handle to default entry.
- **hard_owned** – `True` to treat entries as hard owned.

add_dictionary_var(*owner: str = '0'*, *value: str = ''*) → *DictionaryVar*

Add a new `DictionaryVar` object.

Parameters

- **owner** – handle to owner as hex string.
- **value** – value as string

add_geodata(*owner: str = '0'*, *dxfattribs=None*) → *GeoData*

Creates a new `GeoData` entity and replaces existing ones. The GEODATA entity resides in the OBJECTS section and NOT in the layout entity space, and it is linked to the layout by an extension dictionary located in BLOCK_RECORD of the layout.

The GEODATA entity requires DXF version R2010+. The DXF Reference does not document if other layouts than model space supports geo referencing, so getting/setting geo data may only make sense for the model space layout, but it is also available in paper space layouts.

Parameters

- **owner** – handle to owner as hex string
- **dxfattribs** – DXF attributes for `GeoData` entity

add_image_def(*filename: str*, *size_in_pixel: tuple[int, int]*, *name: str | None = None*) → *ImageDef*

Add an image definition to the objects section.

Add an `ImageDef` entity to the drawing (objects section). *filename* is the image file name as relative or absolute path and *size_in_pixel* is the image size in pixel as (x, y) tuple. To avoid dependencies to external packages, *ezdxf* can not determine the image size by itself. Returns a `ImageDef` entity which is needed to create an image reference. *name* is the internal image name, if set to `None`, name is auto-generated.

Absolute image paths works best for AutoCAD but not really good, you have to update external references manually in AutoCAD, which is not possible in TrueView. If the drawing units differ from 1 meter, you also have to use: `set_raster_variables()`.

Parameters

- **filename** – image file name (absolute path works best for AutoCAD)
- **size_in_pixel** – image size in pixel as (x, y) tuple
- **name** – image name for internal use, None for using filename as name (best for AutoCAD)

add_placeholder(*owner: str = '0'*) → *Placeholder*

Add a new `Placeholder` object.

Parameters

owner – handle to owner as hex string.

add_underlay_def(*filename: str*, *fmt: str = 'pdf'*, *name: str | None = None*) → *UnderlayDefinition*

Add an `UnderlayDefinition` entity to the drawing (OBJECTS section). *filename* is the underlay file name as relative or absolute path and *fmt* as string (pdf, dwf, dgn). The underlay definition is required to create an underlay reference.

Parameters

- **filename** – underlay file name
- **fmt** – file format as string `'pdf'|'dwf'|'dgn'`

- **name** – pdf format = page number to display; dgn format = `'default'`; dwf: ????

add_xrecord(*owner: str = '0'*) → *XRecord*

Add a new `XRecord` object.

Parameters
owner – handle to owner as hex string.

set_raster_variables(*frame: int = 0*, *quality: int = 1*, *units: str = 'm'*) → None

Set raster variables.

Parameters

- **frame** – `0` = do not show image frame; `1` = show image frame
- **quality** – `0` = draft; `1` = high
- **units** – units for inserting images. This defines the real world unit for one drawing unit for the purpose of inserting and scaling images with an associated resolution.

mm	Millimeter
cm	Centimeter
m	Meter (ezdxf default)
km	Kilometer
in	Inch
ft	Foot
yd	Yard
mi	Mile
none	None

(internal API), public interface `set_raster_variables()`

set_wipeout_variables(*frame: int = 0*) → None

Set wipeout variables.

Parameters
frame – `0` = do not show image frame; `1` = show image frame

(internal API)

Tables

Table Classes

Generic Table Class

class ezdxf.sections.table.**Table**

Generic collection of table entries. Table entry names are case insensitive: “Test” == “TEST”.

static key(*name: str*) → str

Unified table entry key.

has_entry(*name: str*) → bool

Returns `True` if a table entry *name* exist.

__contains__(*name: str*) → bool

Returns `True` if a table entry *name* exist.

__len__() → int

Count of table entries.

__iter__() → Iterator[T]

Iterable of all table entries.

new(*name: str*, *dxfattribs=None*) → T

Create a new table entry *name*.

Parameters

- **name** – name of table entry
- **dxfattribs** – additional DXF attributes for table entry

get(*name: str*) → T

Returns table entry *name*.

Parameters
name – name of table entry, case-insensitive

Raises
DXFTableEntryError – table entry does not exist

remove(*name: str*) → None

Removes table entry *name*.

Parameters
name – name of table entry, case-insensitive

Raises
DXFTableEntryError – table entry does not exist

duplicate_entry(*name: str*, *new_name: str*) → T

Returns a new table entry *new_name* as copy of *name*, replaces entry *new_name* if already exist.

Parameters

- **name** – name of table entry, case-insensitive
- **new_name** – name of duplicated table entry

Raises
DXFTableEntryError – table entry does not exist

Layer Table

class ezdxf.sections.table.**LayerTable**

Subclass of *Table*.

Collection of *Layer* objects.

add(*name: str*, ***, *color: int = const.BYLAYER*, *true_color: int | None = None*, *linetype: str = 'Continuous'*, *lineweight: int = const.LINEWEIGHT_BYLAYER*, *plot: bool = True*, *transparency: float | None = None*, *dxfattribs=None*) → Layer

Add a new *Layer*.

Parameters

- **name** (*str*) – layer name
- **color** (*int*) – *AutoCAD Color Index (ACI)* value, default is BYLAYER
- **true_color** (*int*) – true color value, use `ezdxf.rgb2int()` to create `int` values from RGB values
- **linetype** (*str*) – line type name, default is "Continuous"
- **lineweight** (*int*) – line weight, default is BYLAYER
- **plot** (*bool*) – plot layer as bool, default is `True`
- **transparency** – transparency value in the range [0, 1], where 1 is 100% transparent and 0 is opaque
- **dxfattribs** (*dict*) – additional DXF attributes

Linetype Table

class ezdxf.sections.table.**LinetypeTable**

Subclass of *Table*.

Collection of *Linetype* objects.

add(*name: str*, *pattern: Sequence[float] | str*, ***, *description: str = ''*, *length: float = 0.0*, *dxfattribs=None*) → Linetype

Add a new line type entry. The simple line type pattern is a list of floats `[total_pattern_length, elem1, elem2, ...]` where an element > 0 is a line, an element < 0 is a gap and an element == 0.0 is a dot. The definition for complex line types are strings, like: `'A,.5,-.2,["GAS",STANDARD,S=.1,U=0.0,X=-0.1,Y=-.05],-.25'` similar to the line type definitions stored in the line definition *.lin* files, for more information see the tutorial about complex line types. Be aware that not many CAD applications and DXF viewers support complex linetypes.

See also:

- Tutorial for simple line types
- Tutorial for complex line types

Parameters

- **name** (*str*) – line type name
- **pattern** – line type pattern as list of floats or as a string
- **description** (*str*) – line type description, optional
- **length** (*float*) – total pattern length, only for complex line types required
- **dxfattribs** (*dict*) – additional DXF attributes

Style Table

class ezdxf.sections.table.**TextstyleTable**

Subclass of *Table*.

Collection of *Textstyle* objects.

add(*name: str*, *, *font: str*, *dxfattribs=None*) → Textstyle

Add a new text style entry for TTF fonts. The entry must not yet exist, otherwise an `DXFTableEntryError` exception will be raised.

Finding the TTF font files is the task of the DXF viewer and each viewer is different (hint: support files).

Parameters

- **name** (*str*) – text style name
- **font** (*str*) – TTF font file name like "Arial.ttf", the real font file name from the file system is required and only the Windows filesystem is case-insensitive.
- **dxfattribs** (*dict*) – additional DXF attributes

add_shx(*shx_file_name: str*, *, *dxfattribs=None*) → Textstyle

Add a new shape font (SHX file) entry. These are special text style entries and have no name. The entry must not yet exist, otherwise an `DXFTableEntryError` exception will be raised.

Locating the SHX files in the filesystem is the task of the DXF viewer and each viewer is different (hint: support files).

Parameters

- **shx_file_name** (*str*) – shape file name like "gdt.shx"
- **dxfattribs** (*dict*) – additional DXF attributes

get_shx(*shx_file_name: str*) → Textstyle

Get existing entry for a shape file (SHX file), or create a new entry.

Locating the SHX files in the filesystem is the task of the DXF viewer and each viewer is different (hint: support files).

Parameters
shx_file_name (*str*) – shape file name like "gdt.shx"

find_shx(*shx_file_name: str*) → Textstyle | None

Find the shape file (SHX file) text style table entry, by a case-insensitive search.

A shape file table entry has no name, so you have to search by the font attribute.

Parameters
shx_file_name (*str*) – shape file name like "gdt.shx"

discard_shx(*shx_file_name: str*) → None

Discard the shape file (SHX file) text style table entry. Does not raise an exception if the entry does not exist.

Parameters
shx_file_name (*str*) – shape file name like "gdt.shx"

DimStyle Table

class ezdxf.sections.table.**DimStyleTable**

Subclass of *Table*.

Collection of *DimStyle* objects.

add(*name: str*, *, *dxfattribs=None*) → DimStyle

Add a new dimension style table entry.

Parameters

- **name** (*str*) – dimension style name
- **dxfattribs** (*dict*) – DXF attributes

AppID Table

class ezdxf.sections.table.**AppIDTable**

Subclass of *Table*.

Collection of *AppID* objects.

add(*name: str*, *, *dxfattribs=None*) → AppID

Add a new appid table entry.

Parameters

- **name** (*str*) – appid name
- **dxfattribs** (*dict*) – DXF attributes

UCS Table

class ezdxf.sections.table.**UCSTable**

Subclass of *Table*.

Collection of *UCSTableEntry* objects.

add(*name: str*, *, *dxfattribs=None*) → UCSTableEntry

Add a new UCS table entry.

Parameters

- **name** (*str*) – UCS name
- **dxfattribs** (*dict*) – DXF attributes

View Table

class ezdxf.sections.table.**ViewTable**

Subclass of *Table*.

Collection of *View* objects.

add(*name: str*, *, *dxfattribs=None*) → View

Add a new view table entry.

Parameters

- **name** (*str*) – view name
- **dxfattribs** (*dict*) – DXF attributes

Viewport Table

class ezdxf.sections.table.**ViewportTable**

The viewport table stores the modelspace viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport. In contrast to other tables the viewport table can have multiple entries with the same name, because all viewport entries of a multi-viewport configuration are having the same name - the viewport configuration name.

The name of the actual displayed viewport configuration is "*ACTIVE".

Duplication of table entries is not supported: `duplicate_entry()` raises `NotImplementedError`

add(*name: str*, *, *dxfattribs=None*) → VPort

Add a new modelspace viewport entry. A modelspace viewport configuration can consist of multiple viewport entries with the same name.

Parameters

- **name** (*str*) – viewport name, multiple entries possible
- **dxfattribs** (*dict*) – additional DXF attributes

get_config(*self*, *name: str*) → List[*VPort*]

Returns a list of *VPort* objects, for the multi-viewport configuration *name*.

delete_config(*name: str*) → None

Delete all *VPort* objects of the multi-viewport configuration *name*.

Block Record Table

class ezdxf.sections.table.**BlockRecordTable**

Subclass of *Table*.

Collection of *BlockRecord* objects.

add(*name: str*, *, *dxfattribs=None*) → BlockRecord

Add a new block record table entry.

Parameters

- **name** (*str*) – block record name
- **dxfattribs** (*dict*) – DXF attributes

Layer

LAYER (DXF Reference) definition, defines attribute values for entities on this layer for their attributes set to `BYLAYER`.

Important: A layer assignment is just an attribute of a DXF entity, it's not an entity container, the entities are stored in layouts and blocks and the assigned layer is not important for that.

Deleting a layer entry does not delete the entities which reference this layer!

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'LAYER'`
Factory function	`Drawing.layers.new()`

See also:

Basic concepts of *Layers* and *Tutorial for Layers*

class ezdxf.entities.**Layer**

dxf.**handle**

DXF handle (feature for experts)

dxf.**owner**

Handle to owner (*LayerTable*).

dxf.**name**

Layer name, case insensitive and can not contain any of this characters: `<>/\":;?*|=`` (str)

dxf.**flags**

Layer flags (bit-coded values, feature for experts)

1	Layer is frozen; otherwise layer is thawed; use `is_frozen()`, `freeze()` and `thaw()`
2	Layer is frozen by default in new viewports
4	Layer is locked; use `is_locked()`, `lock()`, `unlock()`
16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is for the benefit of AutoCAD commands. It can be ignored by most programs that read DXF files and need not be set by programs that write DXF files)

dxf.**color**

Layer color, but use property *Layer.color* to get/set color value, because color is negative for layer status *off* (int)

dxf.**true_color**

Layer true color value as int, use property *Layer.rgb* to set/get true color value as (r, g, b) tuple.

(requires DXF R2004)

dxf.**linetype**

Name of line type (str)

dxf.**plot**

Plot flag (int). Whether entities belonging to this layer should be drawn when the document is exported (plotted) to pdf. Does not affect visibility inside the CAD application itself.

1	plot layer (default value)
0	don't plot layer

dxf.**lineweight**

Line weight in mm times 100 (e.g. 0.13mm = 13). Smallest line weight is 13 and biggest line weight is 200, values outside this range prevents AutoCAD from loading the file.

`ezdxf.lldxf.const.LINEWEIGHT_DEFAULT` for using global default line weight.

(requires DXF R13)

dxf.**plotstyle_handle**

Handle to plot style name?

(requires DXF R13)

dxf.**material_handle**

Handle to default `Material`.

(requires DXF R13)

rgb

Get/set DXF attribute *dxf.true_color* as (r, g, b) tuple, returns `None` if attribute *dxf.true_color* is not set.

```
layer.rgb = (30, 40, 50)
r, g, b = layer.rgb
```

This is the recommend method to get/set RGB values, when ever possible do not use the DXF low level attribute *dxf.true_color*.

color

Get/set layer color, preferred method for getting the layer color, because *dxf.color* is negative for layer status *off*.

description

Get/set layer description as string

transparency

Get/set layer transparency as float value in the range from 0 to 1. 0 for no transparency (opaque) and 1 for 100% transparency.

is_frozen() → bool

Returns `True` if layer is frozen.

freeze() → None

Freeze layer.

thaw() → None

Thaw layer.

is_locked() → bool

Returns `True` if layer is locked.

lock() → None

Lock layer, entities on this layer are not editable - just important in CAD applications.

unlock() → None

Unlock layer, entities on this layer are editable - just important in CAD applications.

is_off() → bool

Returns `True` if layer is off.

is_on() → bool

Returns `True` if layer is on.

on() → None

Switch layer *on* (visible).

off() → None

Switch layer *off* (invisible).

get_color() → int

Use property *Layer.color* instead.

set_color(*value: int*) → None

Use property *Layer.color* instead.

rename(*name: str*) → None

Rename layer and all known (documented) references to this layer.

> **Warning:** The DXF format is not consistent in storing layer references, the layers are mostly referenced by their case-insensitive name, some later introduced entities do reference layers by handle, which is the safer way in the context of renaming layers.
>
> There is no complete overview of where layer references are stored, third-party entities are black-boxes with unknown content and layer names could be stored in the extended data section of any DXF entity or in XRECORD entities. Which means that in some rare cases references to the old layer name can persist, at least this does not invalidate the DXF document.

Parameters

name – new layer name

Raises

- **ValueError** – *name* contains invalid characters: <>/":;?*|=`
- **ValueError** – layer *name* already exist
- **ValueError** – renaming of layers `'0'` and `'DEFPOINTS'` not possible

get_vp_overrides() → *LayerOverrides*

Returns the *LayerOverrides* object for this layer.

LayerOverrides

class ezdxf.entities.**LayerOverrides**

This object stores the layer attribute overridden in *Viewport* entities, where each *Viewport* can have individual layer attribute overrides.

Layer attributes which can be overridden:

- ACI color
- true color (rgb)
- linetype
- lineweight
- transparency

Get the override object for a certain layer by the *Layer.get_vp_overrides()* method.

It is important to write changes back by calling *commit()*, otherwise the changes are lost.

Important: The implementation of this feature as DXF structures is not documented by the DXF reference, so if you encounter problems or errors, **ALWAYS** provide the DXF files, otherwise it is not possible to help.

---

has_overrides(*vp_handle: str | None = None*) → bool

Returns `True` if attribute overrides exist for the given *Viewport* handle. Returns `True` if *any* attribute overrides exist if the given handle is `None`.

commit() → None

Write *Viewport* overrides back into the *Layer* entity. Without a commit() all changes are lost!

get_color(*vp_handle: str*) → int

Returns the *AutoCAD Color Index (ACI)* override or the original layer value if no override exist.

set_color(*vp_handle: str*, *value: int*) → None

Override the *AutoCAD Color Index (ACI)*.

Raises
ValueError – invalid color value

get_rgb(*vp_handle: str*) → Tuple[int, int, int] | None

Returns the RGB override or the original layer value if no override exist. Returns `None` if no true color value is set.

set_rgb(*vp_handle: str*, *value: Tuple[int, int, int] | None*)

Set the RGB override as (red, gree, blue) tuple or `None` to remove the true color setting.

Raises
ValueError – invalid RGB value

get_transparency(*vp_handle: str*) → float

Returns the transparency override or the original layer value if no override exist. Returns 0.0 for opaque and 1.0 for fully transparent.

set_transparency(*vp_handle: str*, *value: float*) → None

Set the transparency override. A transparency of 0.0 is opaque and 1.0 is fully transparent.

Raises
ValueError – invalid transparency value

get_linetype(*vp_handle: str*) → str

Returns the linetype override or the original layer value if no override exist.

set_linetype(*vp_handle: str*, *value: str*) → None

Set the linetype override.

Raises

ValueError – linetype without a LTYPE table entry

get_lineweight(*vp_handle: str*) → int

Returns the lineweight override or the original layer value if no override exist.

set_lineweight(*vp_handle: str*, *value: int*) → None

Set the lineweight override.

Raises

ValueError – invalid lineweight value

discard(*vp_handle: str | None = None*) → None

Discard all attribute overrides for the given *Viewport* handle or for all *Viewport* entities if the handle is `None`.

Style

Important: DXF is not a layout preserving data format like PDF. It is more similar to the MS Word format. Many applications can open MS Word documents, but the displayed or printed document does not look perfect like the result of MS Word.

The final rendering of DXF files is highly dependent on the interpretation of DXF entities by the rendering engine, and the DXF reference does not provide any guidelines for rendering entities. The biggest visual differences of CAD applications are the text renderings, therefore the only way to get the exact same result is to use the same CAD application.

The DXF format does not and **can not** embed TTF fonts like the PDF format!

The *Textstyle* entity defines a text style (DXF Reference), and can be used by the entities: *Text*, *Attrib*, `Attdef`, *MText*, *Dimension*, *Leader* and *MultiLeader*.

Example to create a new text style "Arial" and to apply this text style:

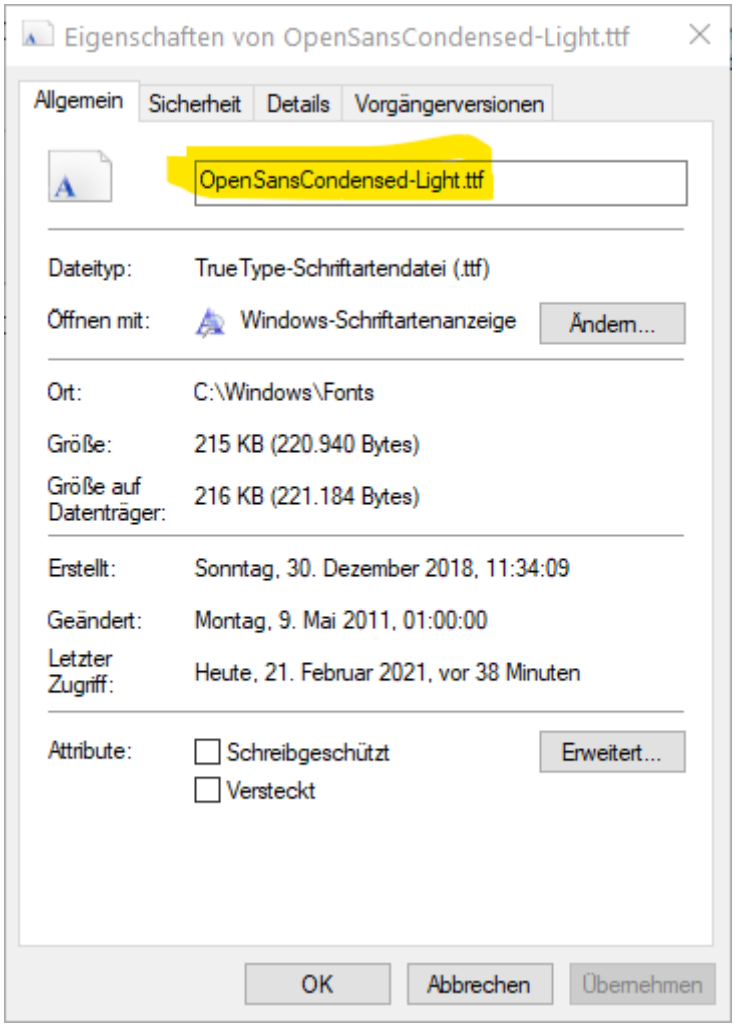
```
doc.styles.add("Arial", font="Arial.ttf")
msp = doc.modelspace()
msp.add_text("my text", dxfattribs={"style": "Arial"})
```

The settings stored in the *Textstyle* entity are the default text style values used by CAD applications if the text settings are not stored in the text entity itself. But not all setting are substituted by the default value. The `height` or `width` attribute must be stored in the text entities itself in order to influence the appearance of the text. It is *recommended* that you do not rely on the default settings in the *Textstyle* entity, set all attributes in the text entity itself if supported.

Font Settings

Just a few settings are available exclusive by the `Textstyle` entity:

The most important setting is the `font` attribute, this attribute defines the rendering font as raw TTF file name, e.g. "Arial.ttf" or "OpenSansCondensed-Light.ttf", this file name is often **not** the name displayed in GUI application and you have to digg down into the fonts folder e.g. ("C:\Windows\Fonts") to get the real file name for the TTF font. Do not include the path!



AutoCAD supports beyond the legacy SHX fonts **only** TTF fonts. The SHX font format is not documented and only available in some CAD applications. The *ezdxf* `drawing` add-on replaces the SHX fonts by TTF fonts, which look similar to the SHX fonts, unfortunately the license of these fonts is unclear, therefore they can not be packaged with *ezdxf*. They are installed automatically if you use an Autodesk product like TrueView, or search the internet at you own risk for these TTF fonts.

The extended font data can provide extra information for the font, it is stored in the XDATA section, not well documented and not widely supported.

Important: The DXF format does not and **can not** embed TTF fonts like the PDF format!

You need to make sure that the CAD application is properly configured to have access to the system fonts. The DXF format has no setting where the CAD application should search for fonts, and does not guarantee that the text rendering

on other computers or operating systems looks the same as on your current system on which you created the DXF.

The second exclusive setting is the vertical text flag in `Textstyle.flags`. The vertical text style is enabled for *all* entities using the text style. Vertical text works only for SHX fonts and is not supported for TTF fonts (in AutoCAD) and is works only for the single line entities *Text* and *Attrib*. Most CAD applications beside AutoCAD and BricsCAD do not support vertical text rendering and even AutoCAD and BricsCAD have problems with vertical text rendering in some circumstances. Using the vertical text feature is not recommended.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'STYLE'`
Factory function	`Drawing.styles.new()`

See also:

Tutorial for Text and DXF internals for *DIMSTYLE Table*.

class ezdxf.entities.**Textstyle**

property is_backward: bool

Get/set text generation flag BACKWARDS, for mirrored text along the x-axis.

property is_upside_down: bool

Get/set text generation flag UPSIDE_DOWN, for mirrored text along the y-axis.

property is_vertical_stacked: bool

Get/set style flag VERTICAL_STACKED, for vertical stacked text.

property is_shape_file: bool

`True` if entry describes a shape.

dxf.**handle**

DXF handle (feature for experts).

dxf.**owner**

Handle to owner (*TextstyleTable*).

dxf.**name**

Style name (str)

dxf.**flags**

Style flags (feature for experts).

1	If set, this entry describes a shape
4	Vertical text
16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)commands. It can be ignored by most programs that read DXF files and need not be set by programs that write DXF files)

dxf.**height**

Fixed height in drawing units as float value, 0 for not fixed.

`dxf.`**`width`**

Width factor as float value, default value is 1.

`dxf.`**`oblique`**

Oblique (slanting) angle in degrees as float value, default value is 0 for no slanting.

`dxf.`**`generation_flags`**

Text generations flags as int value.

2	text is backward (mirrored along the x-axis)
4	text is upside down (mirrored about the base line)

`dxf.`**`last_height`**

Last height used in drawing units as float value.

`dxf.`**`font`**

Raw font file name as string without leading path, e.g. "Arial.ttf" for TTF fonts or the SHX font name like "TXT" or "TXT.SHX".

`dxf.`**`bigfont`**

Big font name as string, blank if none. No documentation how to use this feature, maybe just a legacy artifact.

`property has_extended_font_data: bool`

Returns `True` if extended font data is present.

`get_extended_font_data``()` → tuple[str, bool, bool]

Returns extended font data as tuple (font-family, italic-flag, bold-flag).

The extended font data is optional and not reliable! Returns ("", `False`, `False`) if extended font data is not present.

`set_extended_font_data`(*family: str = ''*, ***, *italic=False*, *bold=False*) → None

Set extended font data, the font-family name *family* is not validated by *ezdxf*. Overwrites existing data.

`discard_extended_font_data``()`

Discard extended font data.

`make_font`(*cap_height: float | None = None*, *width_factor: float | None = None*) → *AbstractFont*

Returns a font abstraction *`AbstractFont`* for this text style. Returns a font for a cap height of 1, if the text style has auto height (*`Textstyle.dxf.height`* is 0) and the given *cap_height* is `None` or 0. Uses the *`Textstyle.dxf.width`* attribute if the given *width_factor* is `None` or 0, the default value is 1. The attribute `Textstyle.dxf.big_font` is ignored.

Linetype

Defines a linetype (DXF Reference).

Subclass of	*`ezdxf.entities.DXFEntity`*
DXF type	`'LTYPE'`
Factory function	`Drawing.linetypes.new()`

See also:

Tutorial for Creating Linetype Pattern

DXF Internals: *LTYPE Table*

class ezdxf.entities.**Linetype**

dxf.**name**

Linetype name (str).

dxf.**owner**

Handle to owner (*Table*).

dxf.**description**

Linetype description (str).

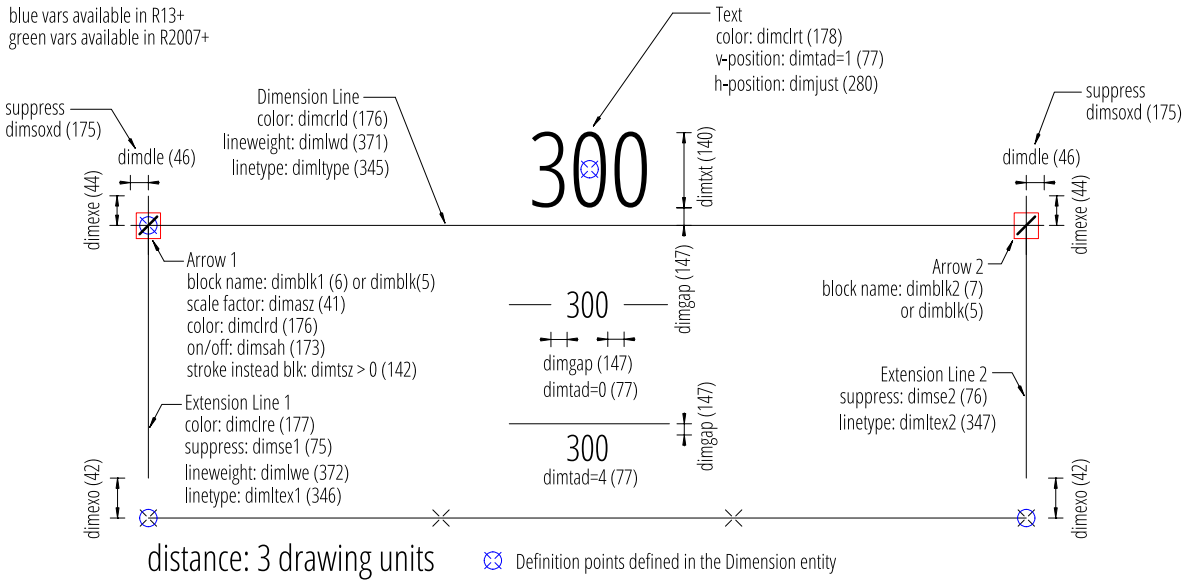
dxf.**length**

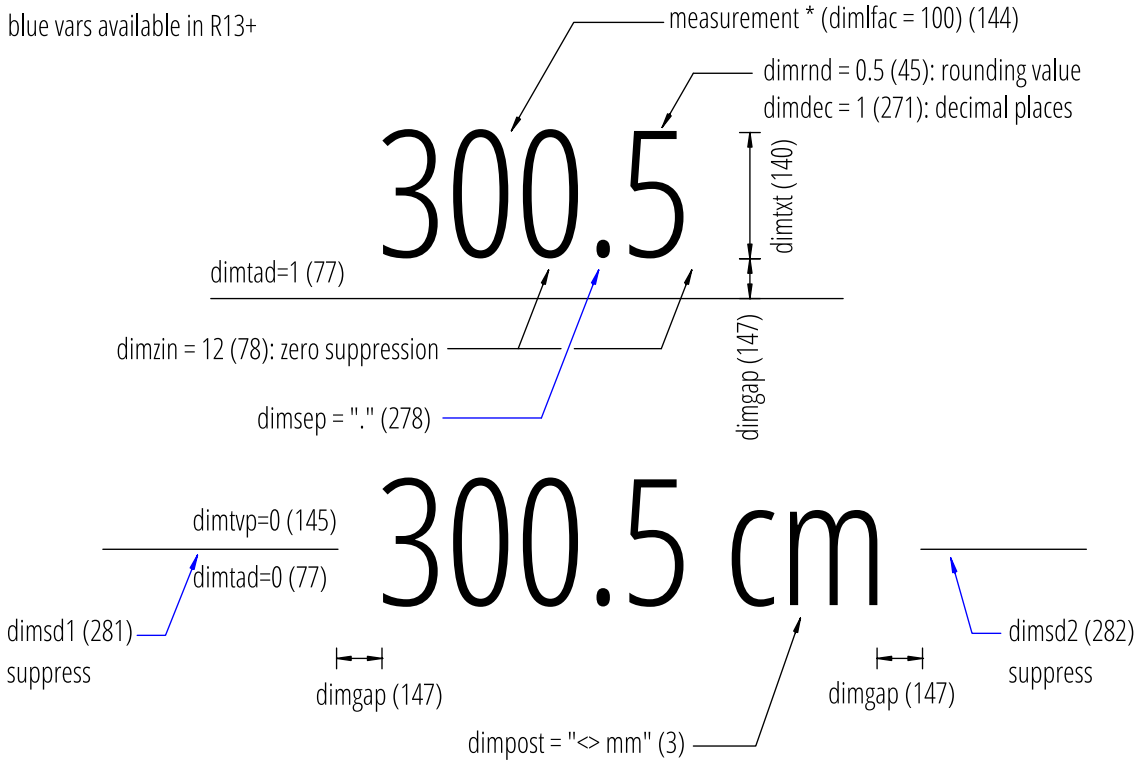
Total pattern length in drawing units (float).

dxf.**items**

Number of linetype elements (int).

DimStyle





DIMSTYLE (DXF Reference) defines the appearance of *Dimension* entities. Each of this dimension variables starting with `"dim..."` can be overridden for any *Dimension* entity individually.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'DIMSTYLE'`
Factory function	`Drawing.dimstyles.new()`

class ezdxf.entities.**DimStyle**

dxf.**owner**

Handle to owner (*Table*).

dxf.**name**

Dimension style name.

dxf.**flags**

Standard flag values (bit-coded values):

16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent XREF has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)

dxf.**dimpost**

Prefix/suffix for primary units dimension values.

dxf.**dimapost**
Prefix/suffix for alternate units dimensions.

dxf.**dimblk**
Block type to use for both arrowheads as name string.

dxf.**dimblk1**
Block type to use for first arrowhead as name string.

dxf.**dimblk2**
Block type to use for second arrowhead as name string.

dxf.**dimscale**
Global dimension feature scale factor. (default=1)

dxf.**dimasz**
Dimension line and arrowhead size. (default=0.25)

dxf.**dimexo**
Distance from origin points to extension lines. (default imperial=0.0625, default metric=0.625)

dxf.**dimdli**
Incremental spacing between baseline dimensions. (default imperial=0.38, default metric=3.75)

dxf.**dimexe**
Extension line distance beyond dimension line. (default imperial=0.28, default metric=2.25)

dxf.**dimrnd**
Rounding value for decimal dimensions. (default=0)

Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.

dxf.**dimdle**
Dimension line extension beyond extension lines. (default=0)

dxf.**dimtp**
Upper tolerance value for tolerance dimensions. (default=0)

dxf.**dimtm**
Lower tolerance value for tolerance dimensions. (default=0)

dxf.**dimtxt**
Size of dimension text. (default imperial=0.28, default metric=2.5)

dxf.**dimcen**
Controls placement of center marks or centerlines. (default imperial=0.09, default metric=2.5)

dxf.**dimtsz**
Controls size of dimension line tick marks drawn instead of arrowheads. (default=0)

dxf.**dimaltf**
Alternate units dimension scale factor. (default=25.4)

dxf.**dimlfac**
Scale factor for linear dimension values. (default=1)

dxf.**dimtvp**
Vertical position of text above or below dimension line if *dimtad* is 0. (default=0)

dxf.**dimtfac**

Scale factor for fractional or tolerance text size. (default=1)

dxf.**dimgap**

Gap size between dimension line and dimension text. (default imperial=0.09, default metric=0.625)

dxf.**dimaltrnd**

Rounding value for alternate dimension units. (default=0)

dxf.**dimtol**

Toggles creation of appended tolerance dimensions. (default imperial=1, default metric=0)

dxf.**dimlim**

Toggles creation of limits-style dimension text. (default=0)

dxf.**dimtih**

Orientation of text inside extension lines. (default imperial=1, default metric=0)

dxf.**dimtoh**

Orientation of text outside extension lines. (default imperial=1, default metric=0)

dxf.**dimse1**

Toggles suppression of first extension line. (default=0)

dxf.**dimse2**

Toggles suppression of second extension line. (default=0)

dxf.**dimtad**

Sets vertical text placement relative to dimension line. (default imperial=0, default metric=1)

0	center
1	above
2	outside, handled like above by *ezdxf*
3	JIS, handled like above by *ezdxf*
4	below

dxf.**dimzin**

Zero suppression for primary units dimensions. (default imperial=0, default metric=8)

Values 0-3 affect feet-and-inch dimensions only.

0	Suppresses zero feet and precisely zero inches
1	Includes zero feet and precisely zero inches
2	Includes zero feet and suppresses zero inches
3	Includes zero inches and suppresses zero feet
4	Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)
8	Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)
12	Suppresses both leading and trailing zeros (for example, 0.5000 becomes .5)

dxf.**dimazin**

Controls zero suppression for angular dimensions. (default=0)

0	Displays all leading and trailing zeros
1	Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)
2	Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)
3	Suppresses leading and trailing zeros (for example, 0.5000 becomes .5)

`dxf.`**`dimalt`**

Enables or disables alternate units dimensioning. (default=0)

`dxf.`**`dimaltd`**

Controls decimal places for alternate units dimensions. (default imperial=2, default metric=3)

`dxf.`**`dimtofl`**

Toggles forced dimension line creation. (default imperial=0, default metric=1)

`dxf.`**`dimsah`**

Toggles appearance of arrowhead blocks. (default=0)

`dxf.`**`dimtix`**

Toggles forced placement of text between extension lines. (default=0)

`dxf.`**`dimsoxd`**

Suppresses dimension lines outside extension lines. (default=0)

`dxf.`**`dimclrd`**

Dimension line, arrowhead, and leader line color. (default=0)

`dxf.`**`dimclre`**

Dimension extension line color. (default=0)

`dxf.`**`dimclrt`**

Dimension text color. (default=0)

`dxf.`**`dimadec`**

Controls the number of decimal places for angular dimensions.

`dxf.`**`dimunit`**

Obsolete, now use DIMLUNIT AND DIMFRAC

`dxf.`**`dimdec`**

Decimal places for dimension values. (default imperial=4, default metric=2)

`dxf.`**`dimtdec`**

Decimal places for primary units tolerance values. (default imperial=4, default metric=2)

`dxf.`**`dimaltu`**

Units format for alternate units dimensions. (default=2)

`dxf.`**`dimalttd`**

Decimal places for alternate units tolerance values. (default imperial=4, default metric=2)

`dxf.`**`dimaunit`**

Unit format for angular dimension values. (default=0)

`dxf.`**`dimfrac`**

Controls the fraction format used for architectural and fractional dimensions. (default=0)

dxf.**dimlunit**

Specifies units for all nonangular dimensions. (default=2)

dxf.**dimdsep**

Specifies a single character to use as a decimal separator. (default imperial = ".", default metric = ",") This is an integer value, use `ord(".")` to write value.

dxf.**dimtmove**

Controls the format of dimension text when it is moved. (default=0)

0	Moves the dimension line with dimension text
1	Adds a leader when dimension text is moved
2	Allows text to be moved freely without a leader

dxf.**dimjust**

Horizontal justification of dimension text. (default=0)

0	Center of dimension line
1	Left side of the dimension line, near first extension line
2	Right side of the dimension line, near second extension line
3	Over first extension line
4	Over second extension line

dxf.**dimsd1**

Toggles suppression of first dimension line. (default=0)

dxf.**dimsd2**

Toggles suppression of second dimension line. (default=0)

dxf.**dimtolj**

Vertical justification for dimension tolerance text. (default=1)

0	Align with bottom line of dimension text
1	Align vertical centered to dimension text
2	Align with top line of dimension text

dxf.**dimtzin**

Zero suppression for tolerances values, see *DimStyle.dxf.dimzin*

dxf.**dimaltz**

Zero suppression for alternate units dimension values. (default=0)

dxf.**dimalttz**

Zero suppression for alternate units tolerance values. (default=0)

dxf.**dimfit**

Obsolete, now use DIMATFIT and DIMTMOVE

dxf.**dimupt**

Controls user placement of dimension line and text. (default=0)

dxf.**dimatfit**

Controls placement of text and arrowheads when there is insufficient space between the extension lines. (default=3)

dxf.**dimtxsty**

Text style used for dimension text by name.

dxf.**dimtxsty_handle**

Text style used for dimension text by handle of STYLE entry. (use *DimStyle.dxf.dimtxsty* to get/set text style by name)

dxf.**dimldrblk**

Specify arrowhead used for leaders by name.

dxf.**dimldrblk_handle**

Specify arrowhead used for leaders by handle of referenced block. (use *DimStyle.dxf.dimldrblk* to get/set arrowhead by name)

dxf.**dimblk_handle**

Block type to use for both arrowheads, handle of referenced block. (use *DimStyle.dxf.dimblk* to get/set arrowheads by name)

dxf.**dimblk1_handle**

Block type to use for first arrowhead, handle of referenced block. (use *DimStyle.dxf.dimblk1* to get/set arrowhead by name)

dxf.**dimblk2_handle**

Block type to use for second arrowhead, handle of referenced block. (use *DimStyle.dxf.dimblk2* to get/set arrowhead by name)

dxf.**dimlwd**

Lineweight value for dimension lines. (default=-2, BYBLOCK)

dxf.**dimlwe**

Lineweight value for extension lines. (default=-2, BYBLOCK)

dxf.**dimltype**

Specifies the linetype used for the dimension line as linetype name, requires DXF R2007+

dxf.**dimltype_handle**

Specifies the linetype used for the dimension line as handle to LTYPE entry, requires DXF R2007+ (use *DimStyle.dxf.dimltype* to get/set linetype by name)

dxf.**dimltex1**

Specifies the linetype used for the extension line 1 as linetype name, requires DXF R2007+

dxf.**dimlex1_handle**

Specifies the linetype used for the extension line 1 as handle to LTYPE entry, requires DXF R2007+ (use *DimStyle.dxf.dimltex1* to get/set linetype by name)

dxf.**dimltex2**

Specifies the linetype used for the extension line 2 as linetype name, requires DXF R2007+

dxf.**dimlex2_handle**

Specifies the linetype used for the extension line 2 as handle to LTYPE entry, requires DXF R2007+ (use *DimStyle.dxf.dimltex2* to get/set linetype by name)

`dxf.`**`dimfxlon`**

Extension line has fixed length if set to 1, requires DXF R2007+

`dxf.`**`dimfxl`**

Length of extension line below dimension line if fixed (`DimStyle.dxf.dimtfxlon` == 1), `DimStyle.dxf.dimexen` defines the length above the dimension line, requires DXF R2007+

`dxf.`**`dimtfill`**

Text fill 0=off; 1=background color; 2=custom color (see *DimStyle.dxf.dimtfillclr*), requires DXF R2007+

`dxf.`**`dimtfillclr`**

Text fill custom color as color index (1-255), requires DXF R2007+

`dxf.`**`dimarcsym`**

Display arc symbol, supported only by *ArcDimension*:

0	arc symbol preceding the measurement text
1	arc symbol above the measurement text
2	disable arc symbol

`copy_to_header`(*doc:* Drawing)

Copy all dimension style variables to HEADER section of *doc*.

`set_arrows`(*blk: str = ''*, *blk1: str = ''*, *blk2: str = ''*, *ldrblk: str = ''*) → None

Set arrows by block names or AutoCAD standard arrow names, set DIMTSZ to `0` which disables tick.

Parameters

- **`blk`** – block/arrow name for both arrows, if DIMSAH is 0
- **`blk1`** – block/arrow name for first arrow, if DIMSAH is 1
- **`blk2`** – block/arrow name for second arrow, if DIMSAH is 1
- **`ldrblk`** – block/arrow name for leader

`set_tick`(*size: float = 1*) → None

Set tick *size*, which also disables arrows, a tick is just an oblique stroke as marker.

Parameters

`size` – arrow size in drawing units

`set_text_align`(*halign: str | None = None*, *valign: str | None = None*, *vshift: float | None = None*) → None

Set measurement text alignment, *halign* defines the horizontal alignment (requires DXF R2000+), *valign* defines the vertical alignment, *above1* and *above2* means above extension line 1 or 2 and aligned with extension line.

Parameters

- **`halign`** – “left”, “right”, “center”, “above1”, “above2”, requires DXF R2000+
- **`valign`** – “above”, “center”, “below”
- **`vshift`** – vertical text shift, if *valign* is “center”; >0 shift upward, <0 shift downwards

`set_text_format`(*prefix: str = ''*, *postfix: str = ''*, *rnd: float | None = None*, *dec: int | None = None*, *sep: str | None = None*, *leading_zeros: bool = True*, *trailing_zeros: bool = True*)

Set dimension text format, like prefix and postfix string, rounding rule and number of decimal places.

Parameters

- **prefix** – Dimension text prefix text as string
- **postfix** – Dimension text postfix text as string
- **rnd** – Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.
- **dec** – Sets the number of decimal places displayed for the primary units of a dimension, requires DXF R2000+
- **sep** – "." or "," as decimal separator, requires DXF R2000+
- **leading_zeros** – Suppress leading zeros for decimal dimensions if `False`
- **trailing_zeros** – Suppress trailing zeros for decimal dimensions if `False`

set_dimline_format (*color: int | None = None*, *linetype: str | None = None*, *lineweight: int | None = None*, *extension: float | None = None*, *disable1: bool | None = None*, *disable2: bool | None = None*)

Set dimension line properties

Parameters

- **color** – color index
- **linetype** – linetype as string, requires DXF R2007+
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm, requires DXF R2000+
- **extension** – extension length
- **disable1** – `True` to suppress first part of dimension line, requires DXF R2000+
- **disable2** – `True` to suppress second part of dimension line, requires DXF R2000+

set_extline_format (*color: int | None = None*, *lineweight: int | None = None*, *extension: float | None = None*, *offset: float | None = None*, *fixed_length: float | None = None*)

Set common extension line attributes.

Parameters

- **color** – color index
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length above dimension line
- **offset** – offset from measurement point
- **fixed_length** – set fixed length extension line, length below the dimension line

set_extline1 (*linetype: str | None = None*, *disable=False*)

Set extension line 1 attributes.

Parameters

- **linetype** – linetype for extension line 1, requires DXF R2007+
- **disable** – disable extension line 1 if `True`

set_extline2 (*linetype: str | None = None*, *disable=False*)

Set extension line 2 attributes.

Parameters

- **linetype** – linetype for extension line 2, requires DXF R2007+
- **disable** – disable extension line 2 if `True`

set_tolerance(*upper: float*, *lower: float | None = None*, *hfactor: float = 1.0*, *align:* MTextLineAlignment | *None = None*, *dec: int | None = None*, *leading_zeros: bool | None = None*, *trailing_zeros: bool | None = None*) → None

Set tolerance text format, upper and lower value, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

- **upper** – upper tolerance value
- **lower** – lower tolerance value, if `None` same as upper
- **hfactor** – tolerance text height factor in relation to the dimension text height
- **align** – tolerance text alignment enum *ezdxf.enums.MTextLineAlignment* requires DXF R2000+
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if `False`, requires DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`, requires DXF R2000+

set_limits(*upper: float*, *lower: float*, *hfactor: float = 1.0*, *dec: int | None = None*, *leading_zeros: bool | None = None*, *trailing_zeros: bool | None = None*) → None

Set limits text format, upper and lower limit values, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

- **upper** – upper limit value added to measurement value
- **lower** – lower limit value subtracted from measurement value
- **hfactor** – limit text height factor in relation to the dimension text height
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if `False`, requires DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`, requires DXF R2000+

VPort

The viewport table (DXF Reference) stores the modelspace viewport configurations. So this entries just modelspace viewports, not paperspace viewports, for paperspace viewports see the *Viewport* entity.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'VPORT'`
Factory function	`Drawing.viewports.new()`

See also:

DXF Internals: *VPORT Configuration Table*

class ezdxf.entities.**VPort**

Subclass of *DXFEntity*

Defines a viewport configurations for the modelspace.

dxf.**owner**

Handle to owner (*ViewportTable*).

dxf.**name**

Viewport name

dxf.**flags**

Standard flag values (bit-coded values):

16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)

dxf.**lower_left**

Lower-left corner of viewport

dxf.**upper_right**

Upper-right corner of viewport

dxf.**center**

View center point (in *DCS*)

dxf.**snap_base**

Snap base point (in *DCS*)

dxf.**snap_spacing**

Snap spacing X and Y

dxf.**grid_spacing**

Grid spacing X and Y

dxf.**direction_point**

View direction from target point (in *WCS*)

dxf.**target_point**

View target point (in *WCS*)

dxf.**height**

View height

dxf.**aspect_ratio**

dxf.**lens_length**

Lens focal length in mm

dxf.**front_clipping**

Front clipping plane (offset from target point)

dxf.**back_clipping**
Back clipping plane (offset from target point)

dxf.**snap_rotation**
Snap rotation angle in degrees

dxf.**view_twist**
View twist angle in degrees

dxf.**status**

dxf.**view_mode**

dxf.**circle_zoom**

dxf.**fast_zoom**

dxf.**ucs_icon**

- bit 0: 0=hide, 1=show
- bit 1: 0=display in lower left corner, 1=display at origin

dxf.**snap_on**

dxf.**grid_on**

dxf.**snap_style**

dxf.**snap_isopair**

reset_wcs() → None
Reset coordinate system to the *WCS*.

View

The View table (DXF Reference) stores named views of the model or paperspace layouts. This stored views makes parts of the drawing or some view points of the model in a CAD applications more accessible. This views have no influence to the drawing content or to the generated output by exporting PDFs or plotting on paper sheets, they are just for the convenience of CAD application users.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	'VIEW'
Factory function	Drawing.views.new()

See also:

DXF Internals: *VIEW Table*

class ezdxf.entities.**View**

dxf.**owner**
Handle to owner (*Table*).

dxf.**name**
Name of view.

`dxf.`**`flags`**

Standard flag values (bit-coded values):

1	If set, this is a paper space view
16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)

`dxf.`**`height`**

View height (in DCS)

`dxf.`**`width`**

View width (in DCS)

`dxf.`**`center_point`**

View center point (in DCS)

`dxf.`**`direction_point`**

View direction from target (in WCS)

`dxf.`**`target_point`**

Target point (in WCS)

`dxf.`**`lens_length`**

Lens length

`dxf.`**`front_clipping`**

Front clipping plane (offset from target point)

`dxf.`**`back_clipping`**

Back clipping plane (offset from target point)

`dxf.`**`view_twist`**

Twist angle in degrees.

`dxf.`**`view_mode`**

View mode (see VIEWMODE system variable)

`dxf.`**`render_mode`**

0	2D Optimized (classic 2D)
1	Wireframe
2	Hidden line
3	Flat shaded
4	Gouraud shaded
5	Flat shaded with wireframe
6	Gouraud shaded with wireframe

`dxf.`**`ucs`**

1 if there is a UCS associated to this view; 0 otherwise

`dxf.`**`ucs_origin`**

UCS origin as (x, y, z) tuple (appears only if `ucs` is set to 1)

dxf.**ucs_xaxis**
: UCS x-axis as (x, y, z) tuple (appears only if `ucs` is set to 1)

dxf.**ucs_yaxis**
: UCS y-axis as (x, y, z) tuple (appears only if `ucs` is set to 1)

dxf.**ucs_ortho_type**
: Orthographic type of UCS (appears only if `ucs` is set to 1)

0	UCS is not orthographic
1	Top
2	Bottom
3	Front
4	Back
5	Left
6	Right

dxf.**elevation**
: UCS elevation

dxf.**ucs_handle**
: Handle of `UCSTable` if UCS is a named UCS. If not present, then UCS is unnamed (appears only if `ucs` is set to 1)

dxf.**base_ucs_handle**
: Handle of `UCSTable` of base UCS if UCS is orthographic. If not present and `ucs_ortho_type` is non-zero, then base UCS is taken to be WORLD (appears only if `ucs` is set to 1)

dxf.**camera_plottable**
: 1 if the camera is plottable

dxf.**background_handle**
: Handle to background object (optional)

dxf.**live_selection_handle**
: Handle to live section object (optional)

dxf.**visual_style_handle**
: Handle to visual style object (optional)

dxf.**sun_handle**
: Sun hard ownership handle.

AppID

Defines an APPID (DXF Reference). These table entries maintain a set of names for all registered applications.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'APPID'`
Factory function	`Drawing.appids.new()`

class ezdxf.entities.**AppID**

dxf.**owner**
: Handle to owner (*Table*).

dxf.**name**
: User-supplied (or application-supplied) application name (for extended data).

dxf.**flags**
: Standard flag values (bit-coded values):

16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)

UCS

Defines an named or unnamed user coordinate system (DXF Reference) for usage in CAD applications. This UCS table entry does not interact with *ezdxf* in any way, to do coordinate transformations by *ezdxf* use the *ezdxf.math.UCS* class.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'UCS'`
Factory function	`Drawing.ucs.new()`

See also:

UCS and *OCS*

class ezdxf.entities.**UCSTableEntry**

dxf.**owner**
: Handle to owner (*Table*).

dxf.**name**
: UCS name (str).

dxf.**flags**
: Standard flags (bit-coded values):

16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)

dxf.**origin**
: Origin as (x, y, z) tuple

dxf.**xaxis**
: X-axis direction as (x, y, z) tuple

dxf.**yaxis**
: Y-axis direction as (x, y, z) tuple

ucs() → *UCS*

Returns an *ezdxf.math.UCS* object for this UCS table entry.

BlockRecord

BLOCK_RECORD (DXF Reference) is the core management structure for *BlockLayout* and *Layout*. This is an internal DXF structure managed by *ezdxf*, package users don't have to care about it.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	'BLOCK_RECORD'
Factory function	Drawing.block_records.new()

class ezdxf.entities.**BlockRecord**

dxf.**owner**

Handle to owner (*Table*).

dxf.**name**

Name of associated BLOCK.

dxf.**layout**

Handle to associated DXFLayout, if paperspace layout or modelspace else "0"

dxf.**explode**

1 for BLOCK references can be exploded else 0

dxf.**scale**

1 for BLOCK references can be scaled else 0

dxf.**units**

BLOCK insert units

0	Unitless
1	Inches
2	Feet
3	Miles
4	Millimeters
5	Centimeters
6	Meters
7	Kilometers
8	Microinches
9	Mils
10	Yards
11	Angstroms
12	Nanometers
13	Microns
14	Decimeters
15	Decameters
16	Hectometers
17	Gigameters
18	Astronomical units
19	Light years
20	Parsecs
21	US Survey Feet
22	US Survey Inch
23	US Survey Yard
24	US Survey Mile

property is_active_paperspace: bool

`True` if is "active" paperspace layout.

property is_any_paperspace: bool

`True` if is any kind of paperspace layout.

property is_any_layout: bool

`True` if is any kind of modelspace or paperspace layout.

property is_block_layout: bool

`True` if not any kind of modelspace or paperspace layout, just a regular block definition.

property is_modelspace: bool

`True` if is the modelspace layout.

property is_xref: bool

`True` if represents an XREF (external reference) or XREF_OVERLAY.

Internal Structure

Do not change this structures, this is just an information for experienced developers!

The BLOCK_RECORD is the owner of all the entities in a layout and stores them in an *EntitySpace* object (`BlockRecord.entity_space`). For each layout exist a BLOCK definition in the BLOCKS section, a reference to the *Block* entity is stored in `BlockRecord.block`.

Modelspace and *Paperspace* layouts require an additional *DXFLayout* object in the OBJECTS section.

See also:

More information about *Block Management Structures* and *Layout Management Structures*.

Blocks

A block definition (*BlockLayout*) is a collection of DXF entities, which can be placed multiply times at different layouts or other blocks as references to the block definition.

See also:

Tutorial for Blocks and DXF Internals: *Block Management Structures*

Block

BLOCK (DXF Reference) entity is embedded into the *BlockLayout* object. The BLOCK entity is accessible by the `BlockLayout.block` attribute.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'BLOCK'`
Factory function	`Drawing.blocks.new()` (returns a *BlockLayout*)

See also:

Tutorial for Blocks and DXF Internals: *Block Management Structures*

class ezdxf.entities.**Block**

dxf.**handle**
: BLOCK handle as plain hex string. (feature for experts)

dxf.**owner**
: Handle to owner as plain hex string. (feature for experts)

dxf.**layer**
: Layer name as string; default value is `'0'`

dxf.**name**
: BLOCK name as string. (case insensitive)

dxf.**base_point**
: BLOCK base point as `(x, y, z)` tuple, default value is `(0, 0, 0)`

  Insertion location referenced by the *Insert* entity to place the block reference and also the center of rotation and scaling.

dxf.**flags**

BLOCK flags (bit-coded)

1	Anonymous block generated by hatching, associative dimensioning, other internal operations, or an application
2	Block has non-constant attribute definitions (this bit is not set if the block has any attribute definitions that are constant, or has no attribute definitions at all)
4	Block is an external reference (xref)
8	Block is an xref overlay
16	Block is externally dependent
32	This is a resolved external reference, or dependent of an external reference (ignored on input)
64	This definition is a referenced external reference (ignored on input)

dxf.**xref_path**

File system path as string, if this block defines an external reference (XREF).

is_layout_block

Returns `True` if this is a *Modelspace* or *Paperspace* block definition.

is_anonymous

Returns `True` if this is an anonymous block generated by hatching, associative dimensioning, other internal operations, or an application.

is_xref

Returns `True` if bock is an external referenced file.

is_xref_overlay

Returns `True` if bock is an external referenced overlay file.

EndBlk

ENDBLK entity is embedded into the *BlockLayout* object. The ENDBLK entity is accessible by the `BlockLayout.endblk` attribute.

Subclass of	*ezdxf.entities.DXFEntity*
DXF type	`'ENDBLK'`

class ezdxf.entities.**EndBlk**

dxf.**handle**

BLOCK handle as plain hex string. (feature for experts)

dxf.**owner**

Handle to owner as plain hex string. (feature for experts)

dxf.**layer**

Layer name as string; should always be the same as *Block.dxf.layer*

Insert

The INSERT entity (DXF Reference) represents a block reference with optional attached attributes as (*Attrib*) entities.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'INSERT'
Factory function	*ezdxf.layouts.BaseLayout.add_blockref()*
Inherited DXF attributes	*Common graphical DXF attributes*

See also:

Tutorial for Blocks

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Insert**

dxf.**name**
BLOCK name (str)

dxf.**insert**
Insertion location of the BLOCK base point as (2D/3D Point in *OCS*)

dxf.**xscale**
Scale factor for x direction (float)

dxf.**yscale**
Scale factor for y direction (float)

Not all CAD applications support non-uniform scaling (e.g. LibreCAD).

dxf.**zscale**
Scale factor for z direction (float)

Not all CAD applications support non-uniform scaling (e.g. LibreCAD).

dxf.**rotation**
Rotation angle in degrees (float)

dxf.**row_count**
Count of repeated insertions in row direction, MINSERT entity if > 1 (int)

dxf.**row_spacing**
Distance between two insert points (MINSERT) in row direction (float)

dxf.**column_count**
Count of repeated insertions in column direction, MINSERT entity if > 1 (int)

dxf.**column_spacing**
Distance between two insert points (MINSERT) in column direction (float)

attribs
A list of all attached *Attrib* entities.

has_scaling

Returns `True` if scaling is applied to any axis.

has_uniform_scaling

Returns `True` if the scale factor is uniform for x-, y- and z-axis, ignoring reflections e.g. (1, 1, -1) is uniform scaling.

mcount

Returns the multi-insert count, MINSERT (multi-insert) processing is required if `mcount` > 1.

set_scale (*factor: float*)

Set a uniform scale factor.

block () → *BlockLayout* | None

Returns the associated `BlockLayout`.

place (*insert:* UVec | *None = None*, *scale: tuple[float, float, float]* | *None = None*, *rotation: float* | *None = None*) → Insert

Set the location, scaling and rotation attributes. Arguments which are `None` will be ignored.

Parameters

- **insert** – insert location as (x, y [,z]) tuple
- **scale** – (x-scale, y-scale, z-scale) tuple
- **rotation** – rotation angle in degrees

grid (*size: tuple[int, int] = (1, 1)*, *spacing: tuple[float, float] = (1, 1)*) → Insert

Place block reference in a grid layout, grid *size* defines the row- and column count, *spacing* defines the distance between two block references.

Parameters

- **size** – grid size as (row_count, column_count) tuple
- **spacing** – distance between placing as (row_spacing, column_spacing) tuple

has_attrib (*tag: str*, *search_const: bool = False*) → bool

Returns `True` if the INSERT entity has an attached ATTRIB entity with the given *tag*. Some applications do not attach constant ATTRIB entities, set *search_const* to `True`, to check for an associated `AttDef` entity with constant content.

Parameters

- **tag** – tag name fo the ATTRIB entity
- **search_const** – search also const ATTDEF entities

get_attrib (*tag: str*, *search_const: bool = False*) → *Attrib* | *AttDef* | None

Get an attached `Attrib` entity with the given *tag*, returns `None` if not found. Some applications do not attach constant ATTRIB entities, set *search_const* to `True`, to get at least the associated `AttDef` entity.

Parameters

- **tag** – tag name of the ATTRIB entity
- **search_const** – search also const ATTDEF entities

get_attrib_text (*tag: str*, *default: str = ''*, *search_const: bool = False*) → str

Get content text of an attached `Attrib` entity with the given *tag*, returns the *default* value if not found. Some applications do not attach constant ATTRIB entities, set *search_const* to `True`, to get content text of the associated `AttDef` entity.

Parameters

- **tag** – tag name of the ATTRIB entity
- **default** – default value if ATTRIB *tag* is absent
- **search_const** – search also const ATTDEF entities

add_attrib(*tag: str*, *text: str*, *insert:* UVec = *(0, 0)*, *dxfattribs=None*) → Attrib

Attach an *Attrib* entity to the block reference.

Example for appending an attribute to an INSERT entity:

```
e.add_attrib('EXAMPLETAG', 'example text').set_placement(
    (3, 7), align=TextEntityAlignment.MIDDLE_CENTER
)
```

Parameters

- **tag** – tag name of the ATTRIB entity
- **text** – content text as string
- **insert** – insert location as (x, y[, z]) tuple in *OCS*
- **dxfattribs** – additional DXF attributes for the ATTRIB entity

add_auto_attribs(*values: dict[str, str]*) → Insert

Attach for each `Attdef` entity, defined in the block definition, automatically an *Attrib* entity to the block reference and set `tag/value` DXF attributes of the ATTRIB entities by the `key/value` pairs (both as strings) of the *values* dict. The ATTRIB entities are placed relative to the insert location of the block reference, which is identical to the block base point.

This method avoids the wrapper block of the *add_auto_blockref()* method, but the visual results may not match the results of CAD applications, especially for non-uniform scaling. If the visual result is very important to you, use the `add_auto_blockref()` method.

Parameters
values – *Attrib* tag values as `tag/value` pairs

delete_attrib(*tag: str*, *ignore=False*) → None

Delete an attached *Attrib* entity from INSERT. Raises an `DXFKeyError` exception, if no ATTRIB for the given *tag* exist if *ignore* is `False`.

Parameters

- **tag** – tag name of the ATTRIB entity
- **ignore** – `False` for raising `DXFKeyError` if ATTRIB *tag* does not exist.

Raises
DXFKeyError – no ATTRIB for the given *tag* exist

delete_all_attribs() → None

Delete all *Attrib* entities attached to the INSERT entity.

transform(*m:* Matrix44) → Insert

Transform INSERT entity by transformation matrix *m* inplace.

Unlike the transformation matrix *m*, the INSERT entity can not represent a non-orthogonal target coordinate system and an `InsertTransformationError` will be raised in that case.

translate(*dx: float*, *dy: float*, *dz: float*) → Insert

Optimized INSERT translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

virtual_entities(*, *skipped_entity_callback: Callable[[DXFGraphic, str], None] | None = None*, *redraw_order=False*) → Iterator[DXFGraphic]

Yields the transformed referenced block content as virtual entities.

This method is meant to examine the block reference entities at the target location without exploding the block reference. These entities are not stored in the entity database, have no handle and are not assigned to any layout. It is possible to convert these entities into regular drawing entities by adding the entities to the entities database and a layout of the same DXF document as the block reference:

```
doc.entitydb.add(entity)
msp = doc.modelspace()
msp.add_entity(entity)
```

Warning: **Non-uniform scale factors** may return incorrect results for some entities (TEXT, MTEXT, ATTRIB).

This method does not resolve the MINSERT attributes, only the sub-entities of the first INSERT will be returned. To resolve MINSERT entities check if multi insert processing is required, that's the case if the property *Insert.mcount* > 1, use the *Insert.multi_insert()* method to resolve the MINSERT entity into multiple INSERT entities.

The *skipped_entity_callback()* will be called for all entities which are not processed, signature: `skipped_entity_callback(entity: DXFEntity, reason: str)`, *entity* is the original (untransformed) DXF entity of the block definition, the *reason* string is an explanation why the entity was skipped.

Parameters

- **skipped_entity_callback** – called whenever the transformation of an entity is not supported and so was skipped
- **redraw_order** – yield entities in ascending redraw order if `True`

multi_insert() → Iterator[Insert]

Yields a virtual INSERT entity for each grid element of a MINSERT entity (multi-insert).

explode(*target_layout:* BaseLayout | *None = None*, *, *redraw_order=False*) → *EntityQuery*

Explodes the block reference entities into the target layout, if target layout is `None`, the layout of the block reference will be used. This method destroys the source block reference entity.

Transforms the block entities into the required *WCS* location by applying the block reference attributes *insert*, *extrusion*, *rotation* and the scale factors *xscale*, *yscale* and *zscale*.

Attached ATTRIB entities are converted to TEXT entities, this is the behavior of the BURST command of the AutoCAD Express Tools.

Warning: **Non-uniform scale factors** may lead to incorrect results some entities (TEXT, MTEXT, ATTRIB).

Parameters

- **target_layout** – target layout for exploded entities, `None` for same layout as source entity.

- **redraw_order** – create entities in ascending redraw order if `True`

Returns
EntityQuery container referencing all exploded DXF entities.

ucs()
Returns the block reference coordinate system as *ezdxf.math.UCS* object.

matrix44() → *Matrix44*
Returns a transformation matrix to transform the block entities from the block reference coordinate system into the *WCS*.

reset_transformation() → None
Reset block reference attributes location, rotation angle and the extrusion vector but preserves the scale factors.

Attrib

The ATTRIB (DXF Reference) entity represents a text value associated with a tag. In most cases an ATTRIB is appended to an *Insert* entity, but it can also be used as a standalone entity.

Subclass of	*ezdxf.entities.Text*
DXF type	`'ATTRIB'`
Factory function	`ezdxf.layouts.BaseLayout.add_attrib()` (stand alone entity)
Factory function	*Insert.add_attrib()* (attached to *Insert*)
Inherited DXF attributes	*Common graphical DXF attributes*

See also:

Tutorial for Blocks

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Attrib**
ATTRIB supports all DXF attributes and methods of parent class *Text*.

dxf.**tag**
Tag to identify the attribute (str)

dxf.**text**
Attribute content as text (str)

property is_invisible: bool
Attribute is invisible if `True`.

property is_const: bool
This is a constant attribute if `True`.

property is_verify: bool
Verification is required on input of this attribute. (interactive CAD application feature)

property is_preset: bool
No prompt during insertion. (interactive CAD application feature)

property has_embedded_mtext_entity: bool

Returns `True` if the entity has an embedded MTEXT entity for multi-line support.

virtual_mtext_entity() → MText

Returns the embedded MTEXT entity as a regular but virtual *MText* entity with the same graphical properties as the host entity.

plain_mtext(*fast=True*) → str

Returns the embedded MTEXT content without formatting codes. Returns an empty string if no embedded MTEXT entity exist.

The *fast* mode is accurate if the DXF content was created by reliable (and newer) CAD applications like AutoCAD or BricsCAD. The *accurate* mode is for some rare cases where the content was created by older CAD applications or unreliable DXF libraries and CAD applications.

The *accurate* mode is **much** slower than the *fast* mode.

Parameters

fast – uses the *fast* mode to extract the plain MTEXT content if `True` or the *accurate* mode if set to `False`

set_mtext(*mtext: MText*, *graphic_properties=True*) → None

Set multi-line properties from a *MText* entity.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- **mtext** – source *MText* entity
- **graphic_properties** – copy graphic properties (color, layer, …) from source MTEXT if `True`

embed_mtext(*mtext: MText*, *graphic_properties=True*) → None

Set multi-line properties from a *MText* entity and destroy the source entity afterwards.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- **mtext** – source *MText* entity
- **graphic_properties** – copy graphic properties (color, layer, …) from source MTEXT if `True`

AttDef

The ATTDEF (DXF Reference) entity is a template in a *BlockLayout*, which will be used to create an attached *Attrib* entity for an *Insert* entity.

Subclass of	*ezdxf.entities.Text*
DXF type	`'ATTDEF'`
Factory function	*ezdxf.layouts.BaseLayout.add_attdef()*
Inherited DXF attributes	*Common graphical DXF attributes*

See also:

Tutorial for Blocks

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**AttDef**

ATTDEF supports all DXF attributes and methods of parent class *Text*.

dxf.**tag**

Tag to identify the attribute (str)

dxf.**text**

Attribute content as text (str)

dxf.**prompt**

Attribute prompt string. (CAD application feature)

dxf.**field_length**

Just relevant to CAD programs for validating user input

property is_invisible: bool

Attribute is invisible if `True`.

property is_const: bool

This is a constant attribute if `True`.

property is_verify: bool

Verification is required on input of this attribute. (interactive CAD application feature)

property is_preset: bool

No prompt during insertion. (interactive CAD application feature)

property has_embedded_mtext_entity: bool

Returns `True` if the entity has an embedded MTEXT entity for multi-line support.

virtual_mtext_entity() → MText

Returns the embedded MTEXT entity as a regular but virtual *MText* entity with the same graphical properties as the host entity.

plain_mtext(*fast=True*) → str

Returns the embedded MTEXT content without formatting codes. Returns an empty string if no embedded MTEXT entity exist.

The *fast* mode is accurate if the DXF content was created by reliable (and newer) CAD applications like AutoCAD or BricsCAD. The *accurate* mode is for some rare cases where the content was created by older CAD applications or unreliable DXF libraries and CAD applications.

The *accurate* mode is **much** slower than the *fast* mode.

Parameters

fast – uses the *fast* mode to extract the plain MTEXT content if `True` or the *accurate* mode if set to `False`

set_mtext(*mtext: MText*, *graphic_properties=True*) → None

Set multi-line properties from a *MText* entity.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- **mtext** – source *MText* entity
- **graphic_properties** – copy graphic properties (color, layer, …) from source MTEXT if `True`

embed_mtext(*mtext: MText*, *graphic_properties=True*) → None

Set multi-line properties from a *MText* entity and destroy the source entity afterwards.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- **mtext** – source *MText* entity
- **graphic_properties** – copy graphic properties (color, layer, …) from source MTEXT if `True`

Layouts

Layout Manager

The layout manager is unique to each DXF drawing, access the layout manager as *layouts* attribute of the *Drawing* object (e.g. `doc.layouts.rename("Layout1", "PlanView")`).

class ezdxf.layouts.**Layouts**

The *Layouts* class manages *Paperspace* layouts and the *Modelspace*.

__len__() → int

Returns count of existing layouts, including the modelspace layout.

__contains__(*name: str*) → bool

Returns `True` if layout *name* exist.

__iter__() → Iterator[Layout]

Returns iterable of all layouts as *Layout* objects, including the modelspace layout.

names() → list[str]

Returns a list of all layout names, all names in original case sensitive form.

names_in_taborder() → list[str]

Returns all layout names in tab order as shown in *CAD* applications.

modelspace() → Modelspace

Returns the *Modelspace* layout.

get(*name: str | None*) → Layout

Returns *Layout* by *name*, case insensitive "Model" == "MODEL".

Parameters

name – layout name as shown in tab, e.g. `'Model'` for modelspace

new(*name: str*, *dxfattribs=None*) → Paperspace

Returns a new *Paperspace* layout.

Parameters

- **name** – layout name as shown in tabs in *CAD* applications

- **dxfattribs** – additional DXF attributes for the `DXFLayout` entity

Raises

- ***DXFValueError*** – Invalid characters in layout name.
- ***DXFValueError*** – Layout *name* already exist.

rename (*old_name: str*, *new_name: str*) → None

Rename a layout from *old_name* to *new_name*. Can not rename layout `'Model'` and the new name of a layout must not exist.

Parameters

- **old_name** – actual layout name, case insensitive
- **new_name** – new layout name, case insensitive

Raises

- ***DXFValueError*** – try to rename `'Model'`
- ***DXFValueError*** – Layout *new_name* already exist.

delete (*name: str*) → None

Delete layout *name* and destroy all entities in that layout.

Parameters
name (*str*) – layout name as shown in tabs

Raises

- ***DXFKeyError*** – if layout *name* do not exists
- ***DXFValueError*** – deleting modelspace layout is not possible
- ***DXFValueError*** – deleting last paperspace layout is not possible

active_layout () → Paperspace

Returns the active paperspace layout.

set_active_layout (*name: str*) → None

Set layout *name* as active paperspace layout.

get_layout_for_entity (*entity: DXFEntity*) → Layout

Returns the owner layout for a DXF *entity*.

Layout Types

A Layout represents and manages DXF entities, there are three different layout objects:

- *Modelspace* is the common working space, containing basic drawing entities.
- *Paperspace* is the arrangement of objects for printing and plotting, this layout contains basic drawing entities and viewports to the *Modelspace*.
- *BlockLayout* works on an associated `Block`, Blocks are collections of DXF entities for reusing by block references.

Warning: Do not instantiate layout classes by yourself - always use the provided factory functions!

Entity Ownership

A layout owns all entities residing in their entity space, therefore the `dxf.owner` attribute of any *DXFGraphic* entity in this layout is the `dxf.handle` of the layout, and deleting an entity from a layout is the end of life of this entity, because it is also deleted from the *EntityDB*. It's possible to just unlink an entity from a layout to assign the entity to another layout, use the *move_to_layout()* method to move entities between layouts.

BaseLayout

class ezdxf.layouts.**BaseLayout**

BaseLayout is the common base class for *Layout* and *BlockLayout*.

is_alive

`False` if layout is deleted.

is_active_paperspace

`True` if is active layout.

is_any_paperspace

`True` if is any kind of paperspace layout.

is_modelspace

`True` if is modelspace layout.

is_any_layout

`True` if is any kind of modelspace or paperspace layout.

is_block_layout

`True` if not any kind of modelspace or paperspace layout, just a regular block definition.

units

set drawing units.

Type
Get/Set layout/block drawing units as enum, see also

Type
ref

__len__() → int

Returns count of entities owned by the layout.

__iter__() → Iterator[*DXFGraphic*]

Returns iterable of all drawing entities in this layout.

__getitem__(*index*)

Get entity at *index*.

The underlying data structure for storing entities is organized like a standard Python list, therefore *index* can be any valid list indexing or slicing term, like a single index `layout[-1]` to get the last entity, or an index slice `layout[:10]` to get the first 10 or less entities as `list[DXFGraphic]`.

get_extension_dict() → *ExtensionDict*

Returns the associated extension dictionary, creates a new one if necessary.

delete_entity(*entity:* DXFGraphic) → None

Delete *entity* from layout entity space and the entity database, this destroys the *entity*.

delete_all_entities() → None

Delete all entities from this layout and from entity database, this destroys all entities in this layout.

unlink_entity(*entity:* DXFGraphic) → None

Unlink *entity* from layout but does not delete entity from the entity database, this removes *entity* just from the layout entity space.

purge()

Remove all destroyed entities from the layout entity space.

query(*query: str = '*'*) → *EntityQuery*

Get all DXF entities matching the *Entity Query String*.

groupby(*dxfattrib: str = ''*, *key: KeyFunc | None = None*) → dict

Returns a `dict` of entity lists, where entities are grouped by a *dxfattrib* or a *key* function.

Parameters

- **dxfattrib** – grouping by DXF attribute like `'layer'`
- **key** – key function, which accepts a `DXFGraphic` entity as argument and returns the grouping key of an entity or `None` to ignore the entity. Reason for ignoring: a queried DXF attribute is not supported by entity.

move_to_layout(*entity:* DXFGraphic, *layout:* BaseLayout) → None

Move entity to another layout.

Parameters

- **entity** – DXF entity to move
- **layout** – any layout (modelspace, paperspace, block) from **same** drawing

set_redraw_order(*handles: dict | Iterable[tuple[str, str]]*) → None

If the header variable $SORTENTS *Regen* flag (bit-code value 16) is set, AutoCAD regenerates entities in ascending handles order.

To change redraw order associate a different sort-handle to entities, this redefines the order in which the entities are regenerated. The *handles* argument can be a dict of entity_handle and sort_handle as (k, v) pairs, or an iterable of (entity_handle, sort_handle) tuples.

The sort-handle doesn't have to be unique, some or all entities can share the same sort-handle and a sort-handle can be an existing handle.

The "0" handle can be used, but this sort-handle will be drawn as latest (on top of all other entities) and not as first as expected.

Parameters

handles – iterable or dict of handle associations; an iterable of 2-tuples (entity_handle, sort_handle) or a dict (k, v) association as (entity_handle, sort_handle)

get_redraw_order() → Iterable[tuple[str, str]]

Returns iterable for all existing table entries as (entity_handle, sort_handle) pairs, see also *set_redraw_order()*.

entities_in_redraw_order(*reverse=False*) → Iterable[*DXFGraphic*]

Yields all entities from layout in ascending redraw order or descending redraw order if *reverse* is `True`.

add_entity(*entity:* DXFGraphic) → None

Add an existing `DXFGraphic` entity to a layout, but be sure to unlink (*unlink_entity()*) entity from the previous owner layout. Adding entities from a different DXF drawing is not supported.

add_foreign_entity(*entity:* DXFGraphic, *copy=True*) → None

Add a foreign DXF entity to a layout, this foreign entity could be from another DXF document or an entity without an assigned DXF document. The intention of this method is to add **simple** entities from another DXF document or from a DXF iterator, for more complex operations use the *importer* add-on. Especially objects with BLOCK section (INSERT, DIMENSION, MLEADER) or OBJECTS section dependencies (IMAGE, UNDERLAY) can not be supported by this simple method.

Not all DXF types are supported and every dependency or resource reference from another DXF document will be removed except attribute layer will be preserved but only with default attributes like color `7` and linetype `CONTINUOUS` because the layer attribute doesn't need a layer table entry.

If the entity is part of another DXF document, it will be unlinked from this document and its entity database if argument *copy* is `False`, else the entity will be copied. Unassigned entities like from DXF iterators will just be added.

Supported DXF types:

- POINT
- LINE
- CIRCLE
- ARC
- ELLIPSE
- LWPOLYLINE
- SPLINE
- POLYLINE
- 3DFACE
- SOLID
- TRACE
- SHAPE
- MESH
- ATTRIB
- ATTDEF
- TEXT
- MTEXT
- HATCH

Parameters

- **entity** – DXF entity to copy or move
- **copy** – if `True` copy entity from other document else unlink from other document

add_point(*location:* UVec, *dxfattribs=None*) → Point

Add a *Point* entity at *location*.

Parameters

- **location** – 2D/3D point in *WCS*

- **dxfattribs** – additional DXF attributes

add_line(*start:* UVec, *end:* UVec, *dxfattribs=None*) → Line

Add a `Line` entity from *start* to *end*.

Parameters

- **start** – 2D/3D point in *WCS*
- **end** – 2D/3D point in *WCS*
- **dxfattribs** – additional DXF attributes

add_circle(*center:* UVec, *radius: float*, *dxfattribs=None*) → *Circle*

Add a `Circle` entity. This is an 2D element, which can be placed in space by using *OCS*.

Parameters

- **center** – 2D/3D point in *WCS*
- **radius** – circle radius
- **dxfattribs** – additional DXF attributes

add_ellipse(*center:* UVec, *major_axis:* UVec = *(1, 0, 0)*, *ratio: float = 1*, *start_param: float = 0*, *end_param: float = math.tau*, *dxfattribs=None*) → *Ellipse*

Add an `Ellipse` entity, *ratio* is the ratio of minor axis to major axis, *start_param* and *end_param* defines start and end point of the ellipse, a full ellipse goes from 0 to 2π. The ellipse goes from start to end param in *counter-clockwise* direction.

Parameters

- **center** – center of ellipse as 2D/3D point in *WCS*
- **major_axis** – major axis as vector (x, y, z)
- **ratio** – ratio of minor axis to major axis in range +/-[1e-6, 1.0]
- **start_param** – start of ellipse curve
- **end_param** – end param of ellipse curve
- **dxfattribs** – additional DXF attributes

add_arc(*center:* UVec, *radius: float*, *start_angle: float*, *end_angle: float*, *is_counter_clockwise: bool = True*, *dxfattribs=None*) → *Arc*

Add an `Arc` entity. The arc goes from *start_angle* to *end_angle* in counter-clockwise direction by default, set parameter *is_counter_clockwise* to `False` for clockwise orientation.

Parameters

- **center** – center of arc as 2D/3D point in *WCS*
- **radius** – arc radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **is_counter_clockwise** – `False` for clockwise orientation
- **dxfattribs** – additional DXF attributes

add_solid(*points: Iterable[*UVec*]*, *dxfattribs=None*) → *Solid*

Add a `Solid` entity, *points* is an iterable of 3 or 4 points.

Hint: The last two vertices are in reversed order: a square has the vertex order 0-1-3-2

Parameters

- **points** – iterable of 3 or 4 2D/3D points in *WCS*
- **dxfattribs** – additional DXF attributes

add_trace(*points: Iterable[*UVec*]*, *dxfattribs=None*) → *Trace*

Add a `Trace` entity, *points* is an iterable of 3 or 4 points.

Hint: The last two vertices are in reversed order: a square has the vertex order 0-1-3-2

Parameters

- **points** – iterable of 3 or 4 2D/3D points in *WCS*
- **dxfattribs** – additional DXF attributes

add_3dface(*points: Iterable[*UVec*]*, *dxfattribs=None*) → *Face3d*

Add a `3DFace` entity, *points* is an iterable 3 or 4 2D/3D points.

Hint: In contrast to SOLID and TRACE, the last two vertices are in regular order: a square has the vertex order 0-1-2-3

Parameters

- **points** – iterable of 3 or 4 2D/3D points in *WCS*
- **dxfattribs** – additional DXF attributes

add_text(*text: str*, ***, *height: float | None = None*, *rotation: float | None = None*, *dxfattribs=None*) → *Text*

Add a `Text` entity, see also `Textstyle`.

Parameters

- **text** – content string
- **height** – text height in drawing units
- **rotation** – text rotation in degrees
- **dxfattribs** – additional DXF attributes

add_blockref(*name: str*, *insert:* UVec, *dxfattribs=None*) → *Insert*

Add an `Insert` entity.

When inserting a block reference into the modelspace or another block layout with different units, the scaling factor between these units should be applied as scaling attributes (`xscale`, ...) e.g. modelspace in meters and block in centimeters, `xscale` has to be 0.01.

Parameters

- **name** – block name as str
- **insert** – insert location as 2D/3D point in *WCS*
- **dxfattribs** – additional DXF attributes

add_auto_blockref(*name: str*, *insert:* UVec, *values: dict[str, str]*, *dxfattribs=None*) → *Insert*

Add an *Insert* entity. This method adds for each `Attdef` entity, defined in the block definition, automatically an `Attrib` entity to the block reference and set (tag, value) DXF attributes of the ATTRIB entities by the (key, value) pairs (both as strings) of the *values* dict.

The Attrib entities are placed relative to the insert point, which is equal to the block base point.

This method wraps the INSERT and all the ATTRIB entities into an anonymous block, which produces the best visual results, especially for non-uniform scaled block references, because the transformation and scaling is done by the CAD application. But this makes evaluation of block references with attributes more complicated, if you prefer INSERT and ATTRIB entities without a wrapper block use the `add_blockref_with_attribs()` method.

Parameters

- **name** – block name
- **insert** – insert location as 2D/3D point in *WCS*
- **values** – *Attrib* tag values as (tag, value) pairs
- **dxfattribs** – additional DXF attributes

add_attdef(*tag: str*, *insert:* UVec = *(0, 0)*, *text: str = ''*, ***, *height: float | None = None*, *rotation: float | None = None*, *dxfattribs=None*) → *AttDef*

Add an *AttDef* as stand alone DXF entity.

Set position and alignment by the idiom:

```
layout.add_attdef("NAME").set_placement(
    (2, 3), align=TextEntityAlignment.MIDDLE_CENTER
)
```

Parameters

- **tag** – tag name as string
- **insert** – insert location as 2D/3D point in *WCS*
- **text** – tag value as string
- **height** – text height in drawing units
- **rotation** – text rotation in degrees
- **dxfattribs** – additional DXF attributes

add_polyline2d(*points: Iterable[*UVec*]*, *format: str | None = None*, ***, *close: bool = False*, *dxfattribs=None*) → *Polyline*

Add a 2D *Polyline* entity.

Parameters

- **points** – iterable of 2D points in *WCS*
- **close** – `True` for a closed polyline
- **format** – user defined point format like *add_lwpolyline()*, default is `None`

- **dxfattribs** – additional DXF attributes

add_polyline3d(*points: Iterable[UVec]*, ***, *close: bool = False*, *dxfattribs=None*) → *Polyline*

Add a 3D `Polyline` entity.

Parameters

- **points** – iterable of 3D points in *WCS*
- **close** – `True` for a closed polyline
- **dxfattribs** – additional DXF attributes

add_polymesh(*size: tuple[int, int] = (3, 3)*, *dxfattribs=None*) → *Polymesh*

Add a `Polymesh` entity, which is a wrapper class for the POLYLINE entity. A polymesh is a grid of *mcount* x *ncount* vertices and every vertex has its own (x, y, z)-coordinates.

Parameters

- **size** – 2-tuple (*mcount*, *ncount*)
- **dxfattribs** – additional DXF attributes

add_polyface(*dxfattribs=None*) → *Polyface*

Add a `Polyface` entity, which is a wrapper class for the POLYLINE entity.

Parameters
dxfattribs – additional DXF attributes for `Polyline` entity

add_shape(*name: str*, *insert:* UVec *= (0, 0)*, *size: float = 1.0*, *dxfattribs=None*) → *Shape*

Add a `Shape` reference to an external stored shape.

Parameters

- **name** – shape name as string
- **insert** – insert location as 2D/3D point in *WCS*
- **size** – size factor
- **dxfattribs** – additional DXF attributes

add_lwpolyline(*points: Iterable[UVec]*, *format: str = 'xyseb'*, ***, *close: bool = False*, *dxfattribs=None*) → *LWPolyline*

Add a 2D polyline as `LWPolyline` entity. A points are defined as (x, y, [start_width, [end_width, [bulge]]]) tuples, but order can be redefined by the *format* argument. Set *start_width*, *end_width* to 0 to be ignored like (x, y, 0, 0, bulge).

The `LWPolyline` is defined as a single DXF entity and needs less disk space than a `Polyline` entity. (requires DXF R2000)

Format codes:

- `x` = x-coordinate
- `y` = y-coordinate
- `s` = start width
- `e` = end width
- `b` = bulge value
- `v` = (x, y [,z]) tuple (z-axis is ignored)

Parameters

- **points** – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuples
- **format** – user defined point format, default is "xyseb"
- **close** – `True` for a closed polyline
- **dxfattribs** – additional DXF attributes

add_mtext (*text: str*, *dxfattribs=None*) → *MText*

Add a multiline text entity with automatic text wrapping at boundaries as `MText` entity. (requires DXF R2000)

Parameters

- **text** – content string
- **dxfattribs** – additional DXF attributes

add_mtext_static_columns (*content: Iterable[str]*, *width: float*, *gutter_width: float*, *height: float*, *dxfattribs=None*) → *MText*

Add a multiline text entity with static columns as `MText` entity. The content is spread across the columns, the count of content strings determine the count of columns.

This factory method adds automatically a column break `"\N"` at the end of each column text to force a new column. The *height* attribute should be big enough to reserve enough space for the tallest column. Too small values produce valid DXF files, but the visual result will not be as expected. The *height* attribute also defines the total height of the MTEXT entity.

(requires DXF R2000)

Parameters

- **content** – iterable of column content
- **width** – column width
- **gutter_width** – distance between columns
- **height** – max. column height
- **dxfattribs** – additional DXF attributes

add_mtext_dynamic_manual_height_columns (*content: str*, *width: float*, *gutter_width: float*, *heights: Sequence[float]*, *dxfattribs=None*) → *MText*

Add a multiline text entity with dynamic columns as `MText` entity. The content is spread across the columns automatically by the CAD application. The *heights* sequence determine the height of the columns, except for the last column, which always takes the remaining content. The height value for the last column is required but can be 0, because the value is ignored. The count of *heights* also determines the count of columns, and `max(heights)` defines the total height of the MTEXT entity, which may be wrong if the last column requires more space.

This current implementation works best for DXF R2018, because the content is stored as a continuous text in a single MTEXT entity. For DXF versions prior to R2018 the content should be distributed across multiple MTEXT entities (one entity per column), which is not done by *ezdxf*, but the result is correct for advanced DXF viewers and CAD application, which do the MTEXT content distribution completely by itself.

(requires DXF R2000)

Parameters

- **content** – column content as a single string
- **width** – column width

- **gutter_width** – distance between columns
- **heights** – column height for each column
- **dxfattribs** – additional DXF attributes

add_mtext_dynamic_auto_height_columns(*content: str*, *width: float*, *gutter_width: float*, *height: float*, *count: int*, *dxfattribs=None*) → *MText*

Add a multiline text entity with as many columns as needed for the given common fixed *height*. The content is spread across the columns automatically by the CAD application. The *height* argument also defines the total height of the MTEXT entity. To get the correct column *count* requires an **exact** MTEXT rendering like AutoCAD, which is not done by *ezdxf*, therefore passing the expected column *count* is required to calculate the correct total width.

This current implementation works best for DXF R2018, because the content is stored as a continuous text in a single MTEXT entity. For DXF versions prior to R2018 the content should be distributed across multiple MTEXT entities (one entity per column), which is not done by *ezdxf*, but the result is correct for advanced DXF viewers and CAD application, which do the MTEXT content distribution completely by itself.

Because of the current limitations the use of this method is not recommend. This situation may improve in future releases, but the exact rendering of the content will also slow down the processing speed dramatically.

(requires DXF R2000)

Parameters

- **content** – column content as a single string
- **width** – column width
- **gutter_width** – distance between columns
- **height** – max. column height
- **count** – expected column count
- **dxfattribs** – additional DXF attributes

add_ray(*start:* UVec, *unit_vector:* UVec, *dxfattribs=None*) → *Ray*

Add a `Ray` that begins at *start* point and continues to infinity (construction line). (requires DXF R2000)

Parameters

- **start** – location 3D point in *WCS*
- **unit_vector** – 3D vector (x, y, z)
- **dxfattribs** – additional DXF attributes

add_xline(*start:* UVec, *unit_vector:* UVec, *dxfattribs=None*) → *XLine*

Add an infinity `XLine` (construction line). (requires DXF R2000)

Parameters

- **start** – location 3D point in *WCS*
- **unit_vector** – 3D vector (x, y, z)
- **dxfattribs** – additional DXF attributes

add_mline(*vertices: Iterable[*UVec*]* | *None = None*, ***, *close: bool = False*, *dxfattribs=None*) → *MLine*

Add a `MLine` entity

Parameters

- **vertices** – MLINE vertices (in *WCS*)

- **close** – `True` to add a closed MLINE
- **dxfattribs** – additional DXF attributes

add_spline(*fit_points: Iterable[UVec] | None = None*, *degree: int = 3*, *dxfattribs=None*) → Spline

Add a B-spline (*Spline* entity) defined by the given *fit_points* - the control points and knot values are created by the CAD application, therefore it is not predictable how the rendered spline will look like, because for every set of fit points exists an infinite set of B-splines.

If *fit_points* is `None`, an "empty" spline will be created, all data has to be set by the user.

The SPLINE entity requires DXF R2000.

AutoCAD creates a spline through fit points by a global curve interpolation and an unknown method to estimate the direction of the start- and end tangent.

See also:

- *Tutorial for Spline*
- *ezdxf.math.fit_points_to_cad_cv()*

Parameters

- **fit_points** – iterable of fit points as `(x, y[, z])` in *WCS*, creates an empty *Spline* if `None`
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **dxfattribs** – additional DXF attributes

add_cad_spline_control_frame(*fit_points: Iterable[UVec]*, *tangents: Iterable[UVec] | None = None*, *dxfattribs=None*) → Spline

Add a *Spline* entity passing through the given fit points.

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in *WCS*
- **tangents** – start- and end tangent, default is autodetect
- **dxfattribs** – additional DXF attributes

add_spline_control_frame(*fit_points: Iterable[UVec]*, *degree: int = 3*, *method: str = 'chord'*, *dxfattribs=None*) → Spline

Add a *Spline* entity passing through the given *fit_points*, the control points are calculated by a global curve interpolation without start- and end tangent constrains. The new SPLINE entity is defined by control points and not by the fit points, therefore the SPLINE looks always the same, no matter which CAD application renders the SPLINE.

- "uniform": creates a uniform t vector, from 0 to 1 evenly spaced, see uniform method
- "distance", "chord": creates a t vector with values proportional to the fit point distances, see chord length method
- "centripetal", "sqrt_chord": creates a t vector with values proportional to the fit point sqrt(distances), see centripetal method
- "arc": creates a t vector with values proportional to the arc length between fit points.

Use function *add_cad_spline_control_frame()* to create SPLINE entities from fit points similar to CAD application including start- and end tangent constraints.

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in *WCS*
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **method** – calculation method for parameter vector t
- **dxfattribs** – additional DXF attributes

add_open_spline(*control_points: Iterable[*UVec*]*, *degree: int = 3*, *knots: Iterable[float] | None = None*, *dxfattribs=None*) → Spline

Add an open uniform *Spline* defined by *control_points*. (requires DXF R2000)

Open uniform B-splines start and end at your first and last control point.

Parameters

- **control_points** – iterable of 3D points in *WCS*
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **knots** – knot values as iterable of floats
- **dxfattribs** – additional DXF attributes

add_rational_spline(*control_points: Iterable[*UVec*]*, *weights: Sequence[float]*, *degree: int = 3*, *knots: Iterable[float] | None = None*, *dxfattribs=None*) → Spline

Add an open rational uniform *Spline* defined by *control_points*. (requires DXF R2000)

weights has to be an iterable of floats, which defines the influence of the associated control point to the shape of the B-spline, therefore for each control point is one weight value required.

Open rational uniform B-splines start and end at the first and last control point.

Parameters

- **control_points** – iterable of 3D points in *WCS*
- **weights** – weight values as iterable of floats
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **knots** – knot values as iterable of floats
- **dxfattribs** – additional DXF attributes

add_hatch(*color: int = 7*, *dxfattribs=None*) → *Hatch*

Add a *Hatch* entity. (requires DXF R2000)

Parameters

- **color** – fill color as :ref`ACI`, default is 7 (black/white).
- **dxfattribs** – additional DXF attributes

add_helix(*radius: float*, *pitch: float*, *turns: float*, *ccw=True*, *dxfattribs=None*) → *Helix*

Add a *Helix* entity.

The center of the helix is always (0, 0, 0) and the helix axis direction is the +z-axis.

Transform the new HELIX by the *transform()* method to your needs.

Parameters

- **radius** – helix radius
- **pitch** – the height of one complete helix turn
- **turns** – count of turns

- **ccw** – creates a counter-clockwise turning (right-handed) helix if `True`
- **dxfattribs** – additional DXF attributes

add_mpolygon(*color: int = const.BYLAYER*, *fill_color: int | None = None*, *dxfattribs=None*) → *MPolygon*

Add a *MPolygon* entity. (requires DXF R2000)

The MPOLYGON entity is not a core DXF entity and is not supported by every CAD application or DXF library.

DXF version R2004+ is required to use a fill color different from BYLAYER. For R2000 the fill color is always BYLAYER, set any ACI value to create a filled MPOLYGON entity.

Parameters

- **color** – boundary color as *AutoCAD Color Index (ACI)*, default is BYLAYER.
- **fill_color** – fill color as *AutoCAD Color Index (ACI)*, default is `None`
- **dxfattribs** – additional DXF attributes

add_mesh(*dxfattribs=None*) → *Mesh*

Add a *Mesh* entity. (requires DXF R2007)

Parameters
dxfattribs – additional DXF attributes

add_image(*image_def:* ImageDef, *insert:* UVec, *size_in_units: tuple[float, float]*, *rotation: float = 0.0*, *dxfattribs=None*) → *Image*

Add an *Image* entity, requires a *ImageDef* entity, see *Tutorial for Image and ImageDef*. (requires DXF R2000)

Parameters

- **image_def** – required image definition as *ImageDef*
- **insert** – insertion point as 3D point in *WCS*
- **size_in_units** – size as (x, y) tuple in drawing units
- **rotation** – rotation angle around the extrusion axis, default is the z-axis, in degrees
- **dxfattribs** – additional DXF attributes

add_wipeout(*vertices: Iterable[*UVec*]*, *dxfattribs=None*) → *Wipeout*

Add a *ezdxf.entities.Wipeout* entity, the masking area is defined by WCS *vertices*.

This method creates only a 2D entity in the xy-plane of the layout, the z-axis of the input vertices are ignored.

add_underlay(*underlay_def:* UnderlayDefinition, *insert:* UVec *= (0, 0, 0)*, *scale=(1, 1, 1)*, *rotation: float = 0.0*, *dxfattribs=None*) → *Underlay*

Add an *Underlay* entity, requires a *UnderlayDefinition* entity, see *Tutorial for Underlay and UnderlayDefinition*. (requires DXF R2000)

Parameters

- **underlay_def** – required underlay definition as *UnderlayDefinition*
- **insert** – insertion point as 3D point in *WCS*
- **scale** – underlay scaling factor as (x, y, z) tuple or as single value for uniform scaling for x, y and z
- **rotation** – rotation angle around the extrusion axis, default is the z-axis, in degrees
- **dxfattribs** – additional DXF attributes

add_linear_dim(*base:* UVec, *p1:* UVec, *p2:* UVec, *location:* UVec | *None* = *None*, *text: str* = *'<>'*, *angle: float* = *0*, *text_rotation: float* | *None* = *None*, *dimstyle: str* = *'EZDXF'*, *override: dict* | *None* = *None*, *dxfattribs=None*) → DimStyleOverride

Add horizontal, vertical and rotated `Dimension` line. If an `UCS` is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: *Tutorial for Linear Dimensions*

This method returns a `DimStyleOverride` object - to create the necessary dimension geometry, you have to call `render()` manually, this two-step process allows additional processing steps on the `Dimension` entity between creation and rendering.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension will do (in UCS)
- **p1** – measurement point 1 and start point of extension line 1 (in UCS)
- **p2** – measurement point 2 and start point of extension line 2 (in UCS)
- **location** – user defined location for the text midpoint (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (`DimStyle` table entry), default is "EZDXF"
- **angle** – angle from ucs/wcs x-axis to dimension line in degrees
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **override** – `DimStyleOverride` attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: `DimStyleOverride`

add_multi_point_linear_dim(*base:* UVec, *points: Iterable[*UVec*]*, *angle: float* = *0*, *ucs:* UCS | *None* = *None*, *avoid_double_rendering: bool* = *True*, *dimstyle: str* = *'EZDXF'*, *override: dict* | *None* = *None*, *dxfattribs=None*, *discard=False*) → None

Add multiple linear dimensions for iterable *points*. If an `UCS` is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: *Tutorial for Linear Dimensions*

This method sets many design decisions by itself, the necessary geometry will be generated automatically, no required nor possible `render()` call. This method is easy to use, but you get what you get.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension will do (in UCS)
- **points** – iterable of measurement points (in UCS)
- **angle** – angle from ucs/wcs x-axis to dimension line in degrees (0 = horizontal, 90 = vertical)
- **ucs** – user defined coordinate system
- **avoid_double_rendering** – suppresses the first extension line and the first arrow if possible for continued dimension entities
- **dimstyle** – dimension style name (DimStyle table entry), default is "EZDXF"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity
- **discard** – discard rendering result for friendly CAD applications like BricsCAD to get a native and likely better rendering result. (does not work with AutoCAD)

add_aligned_dim(*p1:* UVec, *p2:* UVec, *distance: float*, *text: str = '<>'*, *dimstyle: str = 'EZDXF'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Add linear dimension aligned with measurement points *p1* and *p2*. If an *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: *Tutorial for Linear Dimensions*

This method returns a *DimStyleOverride* object, to create the necessary dimension geometry, you have to call `DimStyleOverride.render()` manually, this two-step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **p1** – measurement point 1 and start point of extension line 1 (in UCS)
- **p2** – measurement point 2 and start point of extension line 2 (in UCS)
- **distance** – distance of dimension line from measurement points
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZDXF"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_radius_dim(*center:* UVec, *mpoint:* UVec | *None = None*, *radius: float | None = None*, *angle: float | None = None*, *, *location:* UVec | *None = None*, *text: str = '<>'*, *dimstyle: str = 'EZ_RADIUS'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Add a radius *Dimension* line. The radius dimension line requires a *center* point and a point *mpoint* on the circle or as an alternative a *radius* and a dimension line *angle* in degrees. See also: *Tutorial for Radius Dimensions*

If a *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two-step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Following render types are supported:

- Default text location outside: text aligned with dimension line; dimension style: "EZ_RADIUS"
- Default text location outside horizontal: "EZ_RADIUS" + dimtoh=1
- Default text location inside: text aligned with dimension line; dimension style: "EZ_RADIUS_INSIDE"
- Default text location inside horizontal: "EZ_RADIUS_INSIDE" + dimtih=1
- User defined text location: argument *location* != `None`, text aligned with dimension line; dimension style: "EZ_RADIUS"
- User defined text location horizontal: argument *location* != `None`, "EZ_RADIUS" + dimtoh=1 for text outside horizontal, "EZ_RADIUS" + dimtih=1 for text inside horizontal

Placing the dimension text at a user defined *location*, overrides the *mpoint* and the *angle* argument, but requires a given *radius* argument. The *location* argument does not define the exact text location, instead it defines the dimension line starting at *center* and the measurement text midpoint projected on this dimension line going through *location*, if text is aligned to the dimension line. If text is horizontal, *location* is the kink point of the dimension line from radial to horizontal direction.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **center** – center point of the circle (in UCS)
- **mpoint** – measurement point on the circle, overrides *angle* and *radius* (in UCS)
- **radius** – radius in drawing units, requires argument *angle*
- **angle** – specify angle of dimension line in degrees, requires argument *radius*
- **location** – user defined dimension text location, overrides *mpoint* and *angle*, but requires *radius* (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_RADIUS"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_radius_dim_2p(*center:* UVec, *mpoint:* UVec, *, *text: str = '<>'*, *dimstyle: str = 'EZ_RADIUS'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create a radius dimension by center point, measurement point on the circle and the measurement text at the default location defined by the associated *dimstyle*, for further information see general method *add_radius_dim()*.

- dimstyle “EZ_RADIUS”: places the dimension text outside
- dimstyle “EZ_RADIUS_INSIDE”: places the dimension text inside

Parameters

- **center** – center point of the circle (in UCS)
- **mpoint** – measurement point on the circle (in UCS)
- **text** – `None` or “<>” the measurement is drawn as text, “ “ (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_RADIUS”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_radius_dim_cra(*center: UVec*, *radius: float*, *angle: float*, ***, *text: str = '<>'*, *dimstyle: str = 'EZ_RADIUS'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create a radius dimension by (c)enter point, (r)adius and (a)ngle, the measurement text is placed at the default location defined by the associated *dimstyle*, for further information see general method *add_radius_dim()*.

- dimstyle “EZ_RADIUS”: places the dimension text outside
- dimstyle “EZ_RADIUS_INSIDE”: places the dimension text inside

Parameters

- **center** – center point of the circle (in UCS)
- **radius** – radius in drawing units
- **angle** – angle of dimension line in degrees
- **text** – `None` or “<>” the measurement is drawn as text, “ “ (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_RADIUS”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_diameter_dim(*center: UVec*, *mpoint: UVec | None = None*, *radius: float | None = None*, *angle: float | None = None*, ***, *location: UVec | None = None*, *text: str = '<>'*, *dimstyle: str = 'EZ_RADIUS'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Add a diameter *Dimension* line. The diameter dimension line requires a *center* point and a point *mpoint* on the circle or as an alternative a *radius* and a dimension line *angle* in degrees.

If an *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two-step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **center** – specifies the center of the circle (in UCS)
- **mpoint** – specifies the measurement point on the circle (in UCS)
- **radius** – specify radius, requires argument *angle*, overrides *p1* argument
- **angle** – specify angle of dimension line in degrees, requires argument *radius*, overrides *p1* argument
- **location** – user defined location for the text midpoint (in UCS)
- **text** – `None` or `"<>"` the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_RADIUS"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_diameter_dim_2p(*p1:* UVec, *p2:* UVec, *text: str = '<>'*, *dimstyle: str = 'EZ_RADIUS'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create a diameter dimension by two points on the circle and the measurement text at the default location defined by the associated *dimstyle*, for further information see general method *add_diameter_dim()*. Center point of the virtual circle is the midpoint between *p1* and *p2*.

- dimstyle "EZ_RADIUS": places the dimension text outside
- dimstyle "EZ_RADIUS_INSIDE": places the dimension text inside

Parameters

- **p1** – first point of the circle (in UCS)
- **p2** – second point on the opposite side of the center point of the circle (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_RADIUS"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_angular_dim_2l(*base:* UVec, *line1: tuple[*UVec*,* UVec*]*, *line2: tuple[*UVec*,* UVec*]*, ***, *location:* UVec *| None = None*, *text: str = '<>'*, *text_rotation: float | None = None*, *dimstyle: str = 'EZ_CURVED'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Add angular *Dimension* from two lines. The measurement is always done from *line1* to *line2* in counter-clockwise orientation. This does not always match the result in CAD applications!

If an *UCS* is used for angular dimension rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two-step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension is valid (in UCS)
- **line1** – specifies start leg of the angle (start point, end point) and determines extension line 1 (in UCS)
- **line2** – specifies end leg of the angle (start point, end point) and determines extension line 2 (in UCS)
- **location** – user defined location for the text midpoint (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_angular_dim_3p(*base:* UVec, *center:* UVec, *p1:* UVec, *p2:* UVec, *, *location:* UVec | *None* = *None*, *text: str* = *'<>'*, *text_rotation: float* | *None* = *None*, *dimstyle: str* = *'EZ_CURVED'*, *override: dict* | *None* = *None*, *dxfattribs=None*) → DimStyleOverride

Add angular *Dimension* from three points (center, p1, p2). The measurement is always done from *p1* to *p2* in counter-clockwise orientation. This does not always match the result in CAD applications!

If an *UCS* is used for angular dimension rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two-step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension is valid (in UCS)

- **center** – specifies the vertex of the angle
- **p1** – specifies start leg of the angle (center -> p1) and end-point of extension line 1 (in UCS)
- **p2** – specifies end leg of the angle (center -> p2) and end-point of extension line 2 (in UCS)
- **location** – user defined location for the text midpoint (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_angular_dim_cra(*center:* UVec, *radius: float*, *start_angle: float*, *end_angle: float*, *distance: float*, ***, *location:* UVec | *None = None*, *text: str = '<>'*, *text_rotation: float* | *None = None*, *dimstyle: str = 'EZ_CURVED'*, *override: dict* | *None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create an angular dimension by (c)enter point, (r)adius and start- and end (a)ngles, the measurement text is placed at the default location defined by the associated *dimstyle*. The measurement is always done from *start_angle* to *end_angle* in counter-clockwise orientation. This does not always match the result in CAD applications! For further information see the more generic factory method *add_angular_dim_3p()*.

Parameters

- **center** – center point of the angle (in UCS)
- **radius** – the distance from *center* to the start of the extension lines in drawing units
- **start_angle** – start angle in degrees (in UCS)
- **end_angle** – end angle in degrees (in UCS)
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for the text midpoint (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_angular_dim_arc(*arc:* ConstructionArc, *distance: float*, ***, *location:* UVec | *None = None*, *text: str = '<>'*, *text_rotation: float* | *None = None*, *dimstyle: str = 'EZ_CURVED'*, *override: dict* | *None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create an angular dimension from a *ConstructionArc*. This construction tool can be created from ARC entities and the tool itself provides various construction class methods. The measurement text is placed at the default location defined by the associated *dimstyle*. The measurement is always done from *start_angle* to *end_angle* of the arc in counter-clockwise orientation. This does not always match the result in CAD applications! For further information see the more generic factory method *add_angular_dim_3p()*.

Parameters

- **arc** – *ConstructionArc*
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for the text midpoint (in UCS)
- **text** – `None` or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_arc_dim_3p(*base:* UVec, *center:* UVec, *p1:* UVec, *p2:* UVec, *, *location:* UVec | *None* = *None*, *text: str* = *'<>'*, *text_rotation: float* | *None* = *None*, *dimstyle: str* = *'EZ_CURVED'*, *override: dict* | *None* = *None*, *dxfattribs=None*) → DimStyleOverride

Add *ArcDimension* from three points (center, p1, p2). Point *p1* defines the radius and the start-angle of the arc, point *p2* only defines the end-angle of the arc.

If an *UCS* is used for arc dimension rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two-step process allows additional processing steps on the *ArcDimension* entity between creation and rendering.

Note: *Ezdxf* does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension is valid (in UCS)
- **center** – specifies the vertex of the angle
- **p1** – specifies the radius (center -> p1) and the star angle of the arc, this is also the start point for the 1st extension line (in UCS)
- **p2** – specifies the end angle of the arc. The start 2nd extension line is defined by this angle and the radius defined by p1 (in UCS)
- **location** – user defined location for the text midpoint (in UCS)

- **text** – None or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_arc_dim_cra(*center:* UVec, *radius: float*, *start_angle: float*, *end_angle: float*, *distance: float*, ***, *location:* UVec | *None = None*, *text: str = '<>'*, *text_rotation: float | None = None*, *dimstyle: str = 'EZ_CURVED'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create an arc dimension by (c)enter point, (r)adius and start- and end (a)ngles, the measurement text is placed at the default location defined by the associated *dimstyle*.

Note: *Ezdxf* does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Parameters

- **center** – center point of the angle (in UCS)
- **radius** – the distance from *center* to the start of the extension lines in drawing units
- **start_angle** – start-angle in degrees (in UCS)
- **end_angle** – end-angle in degrees (in UCS)
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for text midpoint (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_arc_dim_arc(*arc:* ConstructionArc, *distance: float*, ***, *location:* UVec | *None = None*, *text: str = '<>'*, *text_rotation: float | None = None*, *dimstyle: str = 'EZ_CURVED'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut method to create an arc dimension from a *ConstructionArc*. This construction tool can be created from ARC entities and the tool itself provides various construction class methods. The measurement text is placed at the default location defined by the associated *dimstyle*.

Note: *Ezdxf* does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Parameters

- **arc** – *ConstructionArc*
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for the text midpoint (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZ_CURVED"
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_ordinate_dim(*feature_location:* UVec, *offset:* UVec, *dtype: int*, *, *origin:* UVec = *NULLVEC*, *rotation: float = 0.0*, *text: str = '<>'*, *dimstyle: str = 'EZDXF'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Add an ordinate type *Dimension* line. The feature location is defined in the global coordinate system, which is set as render UCS, which is the *WCS* by default.

If an *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two-step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdxf* does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **feature_location** – feature location in the global coordinate system (UCS)
- **offset** – offset vector of leader end point from the feature location in the local coordinate system
- **dtype** – 1 = x-type, 0 = y-type
- **origin** – specifies the origin (0, 0) of the local coordinate system in UCS
- **rotation** – rotation angle of the local coordinate system in degrees
- **text** – None or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZDXF"
- **override** – *DimStyleOverride* attributes

- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_ordinate_x_dim(*feature_location: UVec*, *offset: UVec*, *, *origin: UVec = NULLVEC*, *rotation: float = 0.0*, *text: str = '<>'*, *dimstyle: str = 'EZDXF'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut to add an x-type feature ordinate DIMENSION, for more information see *add_ordinate_dim()*.

add_ordinate_y_dim(*feature_location: UVec*, *offset: UVec*, *, *origin: UVec = NULLVEC*, *rotation: float = 0.0*, *text: str = '<>'*, *dimstyle: str = 'EZDXF'*, *override: dict | None = None*, *dxfattribs=None*) → DimStyleOverride

Shortcut to add a y-type feature ordinate DIMENSION, for more information see *add_ordinate_dim()*.

add_leader(*vertices: Iterable[UVec]*, *dimstyle: str = 'EZDXF'*, *override: dict | None = None*, *dxfattribs=None*) → *Leader*

The *Leader* entity represents an arrow, made up of one or more vertices (or spline fit points) and an arrowhead. The label or other content to which the *Leader* is attached is stored as a separate entity, and is not part of the *Leader* itself. (requires DXF R2000)

Leader shares its styling infrastructure with *Dimension*.

By default a *Leader* without any annotation is created. For creating more fancy leaders and annotations see documentation provided by Autodesk or Demystifying DXF: LEADER and MULTILEADER implementation notes .

Parameters

- **vertices** – leader vertices (in *WCS*)
- **dimstyle** – dimension style name (*DimStyle* table entry), default is "EZDXF"
- **override** – override *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes

add_multileader_mtext(*style: str = 'Standard'*, *dxfattribs=None*) → *MultiLeaderMTextBuilder*

Add a *MultiLeader* entity but returns a *MultiLeaderMTextBuilder*.

add_multileader_block(*style: str = 'Standard'*, *dxfattribs=None*) → *MultiLeaderBlockBuilder*

Add a *MultiLeader* entity but returns a *MultiLeaderBlockBuilder*.

add_body(*dxfattribs=None*) → Body

Add a *Body* entity. (requires DXF R2000 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_region(*dxfattribs=None*) → *Region*

Add a *Region* entity. (requires DXF R2000 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_3dsolid(*dxfattribs=None*) → *Solid3d*

Add a 3DSOLID entity (*Solid3d*). (requires DXF R2000 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_surface(*dxfattribs=None*) → Surface

Add a *Surface* entity. (requires DXF R2007 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_extruded_surface(*dxfattribs=None*) → *ExtrudedSurface*

Add a *ExtrudedSurface* entity. (requires DXF R2007 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_lofted_surface(*dxfattribs=None*) → *LoftedSurface*

Add a *LoftedSurface* entity. (requires DXF R2007 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_revolved_surface(*dxfattribs=None*) → *RevolvedSurface*

Add a *RevolvedSurface* entity. (requires DXF R2007 or later)

The ACIS data has to be set as *SAT* or *SAB*.

add_swept_surface(*dxfattribs=None*) → *SweptSurface*

Add a *SweptSurface* entity. (requires DXF R2007 or later)

The ACIS data has to be set as *SAT* or *SAB*.

Layout

class ezdxf.layouts.**Layout**

Layout is a subclass of *BaseLayout* and common base class of *Modelspace* and *Paperspace*.

name

Layout name as shown in tabs of *CAD* applications.

dxf

Returns the DXF name space attribute of the associated *DXFLayout* object.

This enables direct access to the underlying LAYOUT entity, e.g. `Layout.dxf.layout_flags`

__contains__(*entity:* DXFGraphic | *str*) → bool

Returns `True` if *entity* is stored in this layout.

Parameters

entity – `DXFGraphic` object or handle as hex string

reset_extents(*extmin=(+1e20, +1e20, +1e20)*, *extmax=(-1e20, -1e20, -1e20)*) → None

Reset extents to given values or the AutoCAD default values.

"Drawing extents are the bounds of the area occupied by objects." (Quote Autodesk Knowledge Network)

Parameters

- **extmin** – minimum extents or (+1e20, +1e20, +1e20) as default value
- **extmax** – maximum extents or (-1e20, -1e20, -1e20) as default value

reset_limits(*limmin=None*, *limmax=None*) → None

Reset limits to given values or the AutoCAD default values.

"Sets an invisible rectangular boundary in the drawing area that can limit the grid display and limit clicking or entering point locations." (Quote Autodesk Knowledge Network)

The *Paperspace* class has an additional method *reset_paper_limits()* to deduce the default limits from the paper size settings.

Parameters

- **limmin** – minimum limits or (0, 0) as default
- **limmax** – maximum limits or (paper width, paper height) as default value

set_plot_type(*value: int = 5*) → None

0	last screen display
1	drawing extents
2	drawing limits
3	view specific (defined by `Layout.dxf.plot_view_name`)
4	window specific (defined by `Layout.set_plot_window_limits()`)
5	layout information (default)

Parameters

value – plot type

Raises

DXFValueError – for *value* out of range

set_plot_style(*name: str = 'ezdxf.ctb'*, *show: bool = False*) → None

Set plot style file of type *.ctb*.

Parameters

- **name** – plot style filename
- **show** – show plot style effect in preview? (AutoCAD specific attribute)

set_plot_window(*lower_left: tuple[float, float] = (0, 0)*, *upper_right: tuple[float, float] = (0, 0)*) → None

Set plot window size in (scaled) paper space units.

Parameters

- **lower_left** – lower left corner as 2D point
- **upper_right** – upper right corner as 2D point

plot_viewport_borders(*state: bool = True*) → None

show_plot_styles(*state: bool = True*) → None

plot_centered(*state: bool = True*) → None

plot_hidden(*state: bool = True*) → None

use_standard_scale(*state: bool = True*) → None

use_plot_styles(*state: bool = True*) → None

scale_lineweights(*state: bool = True*) → None

print_lineweights(*state: bool = True*) → None

draw_viewports_first(*state: bool = True*) → None

model_type (*state: bool = True*) → None

update_paper (*state: bool = True*) → None

zoom_to_paper_on_update (*state: bool = True*) → None

plot_flags_initializing (*state: bool = True*) → None

prev_plot_init (*state: bool = True*) → None

set_plot_flags (*flag*, *state: bool = True*) → None

Modelspace

class ezdxf.layouts.**Modelspace**

Modelspace is a subclass of *Layout*.

The modelspace contains the "real" world representation of the drawing subjects in real world units.

name

Name of modelspace is fixed as "Model".

new_geodata (*dxfattribs=None*) → *GeoData*

Creates a new `GeoData` entity and replaces existing ones. The GEODATA entity resides in the OBJECTS section and not in the modelspace, it is linked to the modelspace by an `ExtensionDict` located in BLOCK_RECORD of the modelspace.

The GEODATA entity requires DXF R2010. The DXF reference does not document if other layouts than the modelspace supports geo referencing, so I assume getting/setting geo data may only make sense for the modelspace.

Parameters
dxfattribs – DXF attributes for *GeoData* entity

get_geodata () → *GeoData* | None

Returns the *GeoData* entity associated to the modelspace or `None`.

Paperspace

class ezdxf.layouts.**Paperspace**

Paperspace is a subclass of *Layout*.

Paperspace layouts are used to create different drawing sheets of the modelspace subjects for printing or PDF export.

name

Layout name as shown in tabs of *CAD* applications.

page_setup (*size=(297, 210)*, *margins=(10, 15, 10, 15)*, *units='mm'*, *offset=(0, 0)*, *rotation=0*, *scale=16*, *name='ezdxf'*, *device='DWG to PDF.pc3'*)

Setup plot settings and paper size and reset viewports. All parameters in given *units* (mm or inch).

Reset paper limits, extents and viewports.

Parameters

- **size** – paper size as (width, height) tuple

- **margins** – (top, right, bottom, left) hint: clockwise
- **units** – "mm" or "inch"
- **offset** – plot origin offset is 2D point
- **rotation** – see table Rotation
- **scale** – integer in range [0, 32] defines a standard scale type or as tuple(numerator, denominator) e.g. (1, 50) for scale 1:50
- **name** – paper name prefix "{name}_({width}_x_{height}_{unit})"
- **device** – device .pc3 configuration file or system printer name

int	Rotation
0	no rotation
1	90 degrees counter-clockwise
2	upside-down
3	90 degrees clockwise

viewports() → list[*Viewport*]

Get all VIEWPORT entities defined in this paperspace layout.

main_viewport() → *Viewport* | None

Returns the main viewport of this paper space layout, or `None` if no main viewport exist.

add_viewport(*center:* UVec, *size: tuple[float, float]*, *view_center_point:* UVec, *view_height: float*, *dxfattribs=None*) → *Viewport*

Add a new `Viewport` entity.

reset_viewports() → None

Delete all existing viewports, and create a new main viewport.

reset_main_viewport(*center:* UVec = *None*, *size:* UVec = *None*) → *Viewport*

Reset the main viewport of this paper space layout to the given values, or reset them to the default values, deduced from the paper settings. Creates a new main viewport if none exist.

Ezdxf does not create a main viewport by default, because CAD applications don't require one.

Parameters

- **center** – center of the viewport in paper space units
- **size** – viewport size as (width, height) tuple in paper space units

reset_paper_limits() → None

Set paper limits to default values, all values in paperspace units but without plot scale (?).

get_paper_limits() → tuple[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec2*]

Returns paper limits in plot paper units, relative to the plot origin.

plot origin = lower left corner of printable area + plot origin offset

Returns

tuple (Vec2(x1, y1), Vec2(x2, y2)), lower left corner is (x1, y1), upper right corner is (x2, y2).

BlockLayout

class ezdxf.layouts.**BlockLayout**

BlockLayout is a subclass of *BaseLayout*.

Block layouts are reusable sets of graphical entities, which can be referenced by multiple *Insert* entities. Each reference can be placed, scaled and rotated individually and can have it's own set of DXF *Attrib* entities attached.

property name: str

Get/set the BLOCK name

property block: Block | None

the associated *Block* entity.

property endblk: EndBlk | None

the associated *EndBlk* entity.

property dxf

DXF name space of associated *BlockRecord* table entry.

property can_explode: bool

Set property to `True` to allow exploding block references of this block.

property scale_uniformly: bool

Set property to `True` to allow block references of this block only scale uniformly.

property base_point: *Vec3*

Returns the base point of the block.

__contains__(*entity*) → bool

Returns `True` if block contains *entity*.

Parameters

entity – `DXFGraphic` object or handle as hex string

attdefs() → Iterable[AttDef]

Returns iterable of all `Attdef` entities.

has_attdef(*tag: str*) → bool

Returns `True` if an `Attdef` for *tag* exist.

get_attdef(*tag: str*) → DXFGraphic | None

Returns attached `Attdef` entity by *tag* name.

get_attdef_text(*tag: str*, *default: str = ''*) → str

Returns text content for `Attdef` *tag* as string or returns *default* if no `Attdef` for *tag* exist.

Parameters

- **tag** – name of tag
- **default** – default value if *tag* not exist

Groups

A group is just a bunch of DXF entities tied together. All entities of a group has to be in the same layout (modelspace or any paperspace layout but not block). Groups can be named or unnamed, but in reality an unnamed groups has just a special name like "*Annnn". The name of a group has to be unique in the drawing. Groups are organized in the group table, which is stored as attribute *groups* in the *Drawing* object.

Important: Group entities have to reside in the modelspace or an paperspace layout but not in a block definition!

DXFGroup

class ezdxf.entities.dxfgroups.**DXFGroup**

The group name is not stored in the GROUP entity, it is stored in the *GroupCollection* object.

dxf.**description**

group description (string)

dxf.**unnamed**

1 for unnamed, 0 for named group (int)

dxf.**selectable**

1 for selectable, 0 for not selectable group (int)

__iter__() → Iterator[DXFEntity]

Iterate over all DXF entities in *DXFGroup* as instances of DXFGraphic or inherited (LINE, CIRCLE, ...).

__len__() → int

Returns the count of DXF entities in *DXFGroup*.

__getitem__(*item*)

Returns entities by standard Python indexing and slicing.

__contains__(*item: str | DXFEntity*) → bool

Returns True if item is in *DXFGroup*. *item* has to be a handle string or an object of type DXFEntity or inherited.

handles() → Iterable[str]

Iterable of handles of all DXF entities in *DXFGroup*.

edit_data() → list[ezdxf.entities.dxfentity.DXFEntity]

Context manager which yields all the group entities as standard Python list:

```
with group.edit_data() as data:
   # add new entities to a group
   data.append(modelspace.add_line((0, 0), (3, 0)))
   # remove last entity from a group
   data.pop()
```

set_data(*entities: Iterable[DXFEntity]*) → None

Set *entities* as new group content, entities should be an iterable DXFGraphic or inherited (LINE, CIRCLE, ...). Raises DXFValueError if not all entities be on the same layout (modelspace or any paperspace layout but not block)

extend (*entities: Iterable[DXFEntity]*) → None

Add *entities* to *DXFGroup* without immediate verification!

Validation at DXF export may raise a `DXFStructureError`!

clear () → None

Remove all entities from *DXFGroup*, does not delete any drawing entities referenced by this group.

audit (*auditor: Auditor*) → None

Remove invalid entities from *DXFGroup*.

Invalid entities are:

- deleted entities
- all entities which do not reside in model- or paper space
- all entities if they do not reside in the same layout

GroupCollection

Each *Drawing* has one group table, which is accessible by the attribute *groups*.

class ezdxf.entities.dxfgroups.**GroupCollection**

Manages all *DXFGroup* objects of a *Drawing*.

__len__ ()

Returns the count of DXF groups.

__iter__ ()

Iterate over all existing groups as (*name*, *group*) tuples. *name* is the name of the group as string and *group* is an *DXFGroup* object.

__contains__ ()

Returns `True` if a group *name* exist.

get (*name: str*) → *DXFGroup*

Returns the group *name*. Raises `DXFKeyError` if group *name* does not exist.

groups () → Iterator[*DXFGroup*]

Iterable of all existing groups.

new (*name: str | None = None*, *description: str = ''*, *selectable: bool = True*) → *DXFGroup*

Creates a new group. If *name* is `None` an unnamed group is created, which has an automatically generated name like "*Annnn". Group names are case-insensitive.

Parameters

- **name** – group name as string
- **description** – group description as string
- **selectable** – group is selectable if `True`

delete (*group:* DXFGroup | *str*) → None

Delete *group*, *group* can be an object of type *DXFGroup* or a group name as string.

clear ()

Delete all groups.

audit(*auditor: Auditor*) → None

Removes empty groups and invalid handles from all groups.

DXF Entities

All DXF entities can only reside in the *BaseLayout* and inherited classes like *Modelspace*, *Paperspace* and *BlockLayout*.

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

DXF Entity Base Class

Common base class for all DXF entities and objects.

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**DXFEntity**

dxf

The DXF attributes namespace:

```
# set attribute value
entity.dxf.layer = 'MyLayer'

# get attribute value
linetype = entity.dxf.linetype

# delete attribute
del entity.dxf.linetype
```

dxf.**handle**

DXF *handle* is a unique identifier as plain hex string like `F000`. (feature for experts)

dxf.**owner**

Handle to *owner* as plain hex string like `F000`. (feature for experts)

doc

Get the associated *Drawing* instance.

property is_alive: bool

Is `False` if entity has been deleted.

property is_virtual: bool

Is `True` if entity is a virtual entity.

property is_bound: bool

Is `True` if entity is bound to DXF document.

property is_copy: bool

Is `True` if the entity is a copy.

property uuid: UUID

Returns a UUID, which allows to distinguish even virtual entities without a handle.

Dynamic attribute: this UUID will be created at the first request.

property source_of_copy: DXFEntity | None

The immediate source entity if this entity is a copy else `None`. Never references a destroyed entity.

property origin_of_copy: DXFEntity | None

The origin source entity if this entity is a copy else `None`. References the first non-virtual source entity and never references a destroyed entity.

property has_source_block_reference: bool

Is `True` if this virtual entity was created by a block reference.

property source_block_reference: *Insert* | None

The source block reference (INSERT) which created this virtual entity. The property is `None` if this entity was not created by a block reference.

dxftype() → str

Get DXF type as string, like `LINE` for the line entity.

__str__() → str

Returns a simple string representation.

__repr__() → str

Returns a simple string representation including the class.

has_dxf_attrib(*key: str*) → bool

Returns `True` if DXF attribute *key* really exist.

Raises `DXFAttributeError` if *key* is not an supported DXF attribute.

is_supported_dxf_attrib(*key: str*) → bool

Returns `True` if DXF attrib *key* is supported by this entity. Does not grant that attribute *key* really exist.

get_dxf_attrib(*key: str*, *default: Any = None*) → Any

Get DXF attribute *key*, returns *default* if key doesn't exist, or raise `DXFValueError` if *default* is `DXFValueError` and no DXF default value is defined:

```
layer = entity.get_dxf_attrib("layer")
# same as
layer = entity.dxf.layer
```

Raises `DXFAttributeError` if *key* is not an supported DXF attribute.

set_dxf_attrib(*key: str*, *value: Any*) → None

Set new *value* for DXF attribute *key*:

```
entity.set_dxf_attrib("layer", "MyLayer")
# same as
entity.dxf.layer = "MyLayer"
```

Raises `DXFAttributeError` if *key* is not an supported DXF attribute.

del_dxf_attrib(*key: str*) → None

Delete DXF attribute *key*, does not raise an error if attribute is supported but not present.

Raises `DXFAttributeError` if *key* is not an supported DXF attribute.

dxfattribs (*drop: set[str] | None = None*) → dict

Returns a `dict` with all existing DXF attributes and their values and exclude all DXF attributes listed in set *drop*.

update_dxf_attribs (*dxfattribs: dict*) → None

Set DXF attributes by a `dict` like `{'layer': 'test', 'color': 4}`.

set_flag_state (*flag: int*, *state: bool = True*, *name: str = 'flags'*) → None

Set binary coded *flag* of DXF attribute *name* to 1 (on) if *state* is `True`, set *flag* to 0 (off) if *state* is `False`.

get_flag_state (*flag: int*, *name: str = 'flags'*) → bool

Returns `True` if any *flag* of DXF attribute is 1 (on), else `False`. Always check only one flag state at the time.

has_extension_dict

Returns `True` if entity has an attached *ExtensionDict* instance.

get_extension_dict () → *ExtensionDict*

Returns the existing *ExtensionDict* instance.

Raises
AttributeError – extension dict does not exist

new_extension_dict () → *ExtensionDict*

Create a new *ExtensionDict* instance .

discard_extension_dict () → None

Delete *ExtensionDict* instance .

has_app_data (*appid: str*) → bool

Returns `True` if application defined data for *appid* exist.

get_app_data (*appid: str*) → *Tags*

Returns application defined data for *appid*.

Parameters
appid – application name as defined in the APPID table.

Raises
DXFValueError – no data for *appid* found

set_app_data (*appid: str*, *tags: Iterable*) → None

Set application defined data for *appid* as iterable of tags.

Parameters

- **appid** – application name as defined in the APPID table.
- **tags** – iterable of (code, value) tuples or *DXFTag*

discard_app_data (*appid: str*)

Discard application defined data for *appid*. Does not raise an exception if no data for *appid* exist.

has_xdata (*appid: str*) → bool

Returns `True` if extended data for *appid* exist.

get_xdata (*appid: str*) → *Tags*

Returns extended data for *appid*.

Parameters
appid – application name as defined in the APPID table.

Raises
DXFValueError – no extended data for *appid* found

set_xdata (*appid: str*, *tags: Iterable*) → None

Set extended data for *appid* as iterable of tags.

Parameters

- **appid** – application name as defined in the APPID table.
- **tags** – iterable of (code, value) tuples or *DXFTag*

discard_xdata (*appid: str*) → None

Discard extended data for *appid*. Does not raise an exception if no extended data for *appid* exist.

has_xdata_list (*appid: str*, *name: str*) → bool

Returns `True` if a tag list *name* for extended data *appid* exist.

get_xdata_list (*appid: str*, *name: str*) → *Tags*

Returns tag list *name* for extended data *appid*.

Parameters

- **appid** – application name as defined in the APPID table.
- **name** – extended data list name

Raises
DXFValueError – no extended data for *appid* found or no data list *name* not found

set_xdata_list (*appid: str*, *name: str*, *tags: Iterable*) → None

Set tag list *name* for extended data *appid* as iterable of tags.

Parameters

- **appid** – application name as defined in the APPID table.
- **name** – extended data list name
- **tags** – iterable of (code, value) tuples or *DXFTag*

discard_xdata_list (*appid: str*, *name: str*) → None

Discard tag list *name* for extended data *appid*. Does not raise an exception if no extended data for *appid* or no tag list *name* exist.

replace_xdata_list (*appid: str*, *name: str*, *tags: Iterable*) → None

Replaces tag list *name* for existing extended data *appid* by *tags*. Appends new list if tag list *name* do not exist, but raises `DXFValueError` if extended data *appid* do not exist.

Parameters

- **appid** – application name as defined in the APPID table.
- **name** – extended data list name
- **tags** – iterable of (code, value) tuples or *DXFTag*

Raises
DXFValueError – no extended data for *appid* found

has_reactors () → bool

Returns `True` if entity has reactors.

get_reactors() → list[str]

Returns associated reactors as list of handles.

set_reactors(*handles: Iterable[str]*) → None

Set reactors as list of handles.

append_reactor_handle(*handle: str*) → None

Append *handle* to reactors.

discard_reactor_handle(*handle: str*) → None

Discard *handle* from reactors. Does not raise an exception if *handle* does not exist.

DXF Graphic Entity Base Class

Common base class for all graphical DXF entities.

All graphical entities reside in an entity space like *Modelspace*, any *Paperspace* or *BlockLayout*.

See also:

- *ezdxf.gfxattribs* module, helper tools to set graphical attributes of DXF entities
- *ezdxf.colors* module
- *Tutorial for Common Graphical Attributes*

Subclass of *ezdxf.entities.DXFEntity*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**DXFGraphic**

rgb

Get/set DXF attribute *dxf.true_color* as (r, g, b) tuple, returns `None` if attribute *dxf.true_color* is not set.

```
entity.rgb = (30, 40, 50)
r, g, b = entity.rgb
```

This is the recommend method to get/set RGB values, when ever possible do not use the DXF low level attribute *dxf.true_color*.

transparency

Get/set the transparency value as float. The transparency value is in the range from 0 to 1, where 0 means the entity is opaque and 1 means the entity is 100% transparent (invisible). This is the recommend method to get/set the transparency value, when ever possible do not use the DXF low level attribute *DXFGraphic.dxf.transparency*.

This attribute requires DXF R2004 or later, returns 0 for older DXF versions and raises `DXFAttributeError` for setting transparency in older DXF versions.

property is_transparency_by_layer: bool

Returns `True` if entity inherits transparency from layer.

property is_transparency_by_block: bool

Returns `True` if entity inherits transparency from block.

ocs() → *OCS*

Returns object coordinate system (*OCS*) for 2D entities like *Text* or *Circle*, returns a pass-through OCS for entities without OCS support.

get_layout() → *BaseLayout* | None

Returns the owner layout or returns `None` if entity is not assigned to any layout.

unlink_from_layout() → None

Unlink entity from associated layout. Does nothing if entity is already unlinked.

It is more efficient to call the *unlink_entity()* method of the associated layout, especially if you have to unlink more than one entity.

copy_to_layout(*layout:* BaseLayout) → *DXFEntity*

Copy entity to another *layout*, returns new created entity as *DXFEntity* object. Copying between different DXF drawings is not supported.

Parameters
layout – any layout (model space, paper space, block)

Raises
DXFStructureError – for copying between different DXF drawings

move_to_layout(*layout:* BaseLayout, *source:* BaseLayout | *None* = *None*) → None

Move entity from model space or a paper space layout to another layout. For block layout as source, the block layout has to be specified. Moving between different DXF drawings is not supported.

Parameters

- **layout** – any layout (model space, paper space, block)
- **source** – provide source layout, faster for DXF R12, if entity is in a block layout

Raises
DXFStructureError – for moving between different DXF drawings

graphic_properties() → dict

Returns the important common properties layer, color, linetype, lineweight, ltscale, true_color and color_name as *dxfattribs* dict.

has_hyperlink() → bool

Returns `True` if entity has an attached hyperlink.

get_hyperlink() → tuple[str, str, str]

Returns hyperlink, description and location.

set_hyperlink(*link: str*, *description: str* | *None* = *None*, *location: str* | *None* = *None*)

Set hyperlink of an entity.

transform(*m:* Matrix44) → DXFGraphic

Inplace transformation interface, returns *self* (floating interface).

Parameters
m – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

translate(*dx: float*, *dy: float*, *dz: float*) → DXFGraphic

Translate entity inplace about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).

Basic implementation uses the `transform()` interface, subclasses may have faster implementations.

scale(*sx: float*, *sy: float*, *sz: float*) → DXFGraphic

Scale entity inplace about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).

scale_uniform(*s: float*) → DXFGraphic

Scale entity inplace uniform about *s* in x-axis, y-axis and z-axis, returns *self* (floating interface).

rotate_x(*angle: float*) → DXFGraphic

Rotate entity inplace about x-axis, returns *self* (floating interface).

Parameters
angle – rotation angle in radians

rotate_y(*angle: float*) → DXFGraphic

Rotate entity inplace about y-axis, returns *self* (floating interface).

Parameters
angle – rotation angle in radians

rotate_z(*angle: float*) → DXFGraphic

Rotate entity inplace about z-axis, returns *self* (floating interface).

Parameters
angle – rotation angle in radians

rotate_axis(*axis:* UVec, *angle: float*) → DXFGraphic

Rotate entity inplace about vector *axis*, returns *self* (floating interface).

Parameters

- **axis** – rotation axis as tuple or `Vec3`
- **angle** – rotation angle in radians

Common graphical DXF attributes

`DXFGraphic.dxf.`**layer**

Layer name as string; default = "0"

`DXFGraphic.dxf.`**linetype**

Linetype as string, special names "BYLAYER", "BYBLOCK"; default value is "BYLAYER"

`DXFGraphic.dxf.`**color**

AutoCAD Color Index (ACI), default value is 256

Constants defined in `ezdxf.lldxf.const` or use the `ezdxf.colors` module

0	BYBLOCK
256	BYLAYER
257	BYOBJECT

DXFGraphic.dxf.**lineweight**

Line weight in mm times 100 (e.g. 0.13mm = 13). There are fixed valid lineweights which are accepted by AutoCAD, other values prevents AutoCAD from loading the DXF document, BricsCAD isn't that picky. (requires DXF R2000)

Constants defined in *ezdxf.lldxf.const*

-1	LINEWEIGHT_BYLAYER
-2	LINEWEIGHT_BYBLOCK
-3	LINEWEIGHT_DEFAULT

Valid DXF lineweights stored in VALID_DXF_LINEWEIGHTS: 0, 5, 9, 13, 15, 18, 20, 25, 30, 35, 40, 50, 53, 60, 70, 80, 90, 100, 106, 120, 140, 158, 200, 211

DXFGraphic.dxf.**ltscale**

Line type scale as float; default value is 1.0; (requires DXF R2000)

DXFGraphic.dxf.**invisible**

1 for invisible, 0 for visible; default value is 0; (requires DXF R2000)

DXFGraphic.dxf.**paperspace**

0 for entity resides in modelspace or a block, 1 for paperspace, this attribute is set automatically by adding an entity to a layout (feature for experts); default value is 0

DXFGraphic.dxf.**extrusion**

Extrusion direction as 3D vector; default value is (0, 0, 1)

DXFGraphic.dxf.**thickness**

Entity thickness as float; default value is 0.0; (requires DXF R2000)

DXFGraphic.dxf.**true_color**

True color value as int 0x00RRGGBB, use *DXFGraphic.rgb* to get/set true color values as (r, g, b) tuples. (requires DXF R2004)

DXFGraphic.dxf.**color_name**

Color name as string. (requires DXF R2004)

DXFGraphic.dxf.**transparency**

Transparency value as int, 0x020000TT, 0x00 = 100% transparent / 0xFF = opaque, special value 0x01000000 means transparency by block. An unset transparency value means transparency by layer. Use *DXFGraphic.transparency* to get/set transparency as float value, and the properties *DXFGraphic.is_transparency_by_block* and *DXFGraphic.is_transparency_by_layer* to check special cases.

(requires DXF R2004)

DXFGraphic.dxf.**shadow_mode**

0	casts and receives shadows
1	casts shadows
2	receives shadows
3	ignores shadows

(requires DXF R2007)

See also:

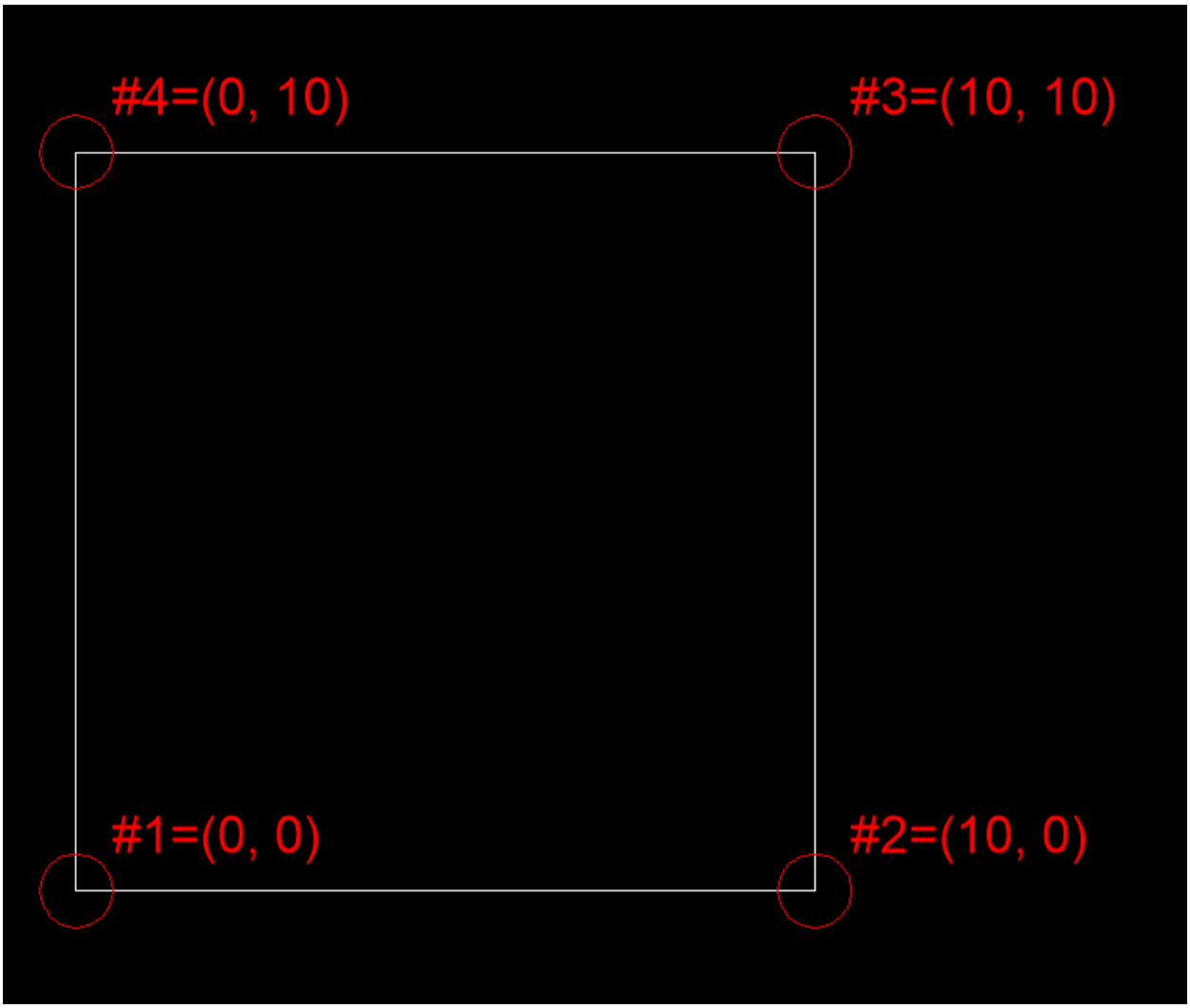
- *ezdxf.gfxattribs* module, helper tools to set graphical attributes of DXF entities
- *ezdxf.colors* module
- *Tutorial for Common Graphical Attributes*

Face3d

The 3DFACE entity (DXF Reference) is real 3D solid filled triangle or quadrilateral. Access vertices by name (`entity.dxf.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`).

Unlike the entities *Solid* and *Trace*, the vertices of *Face3d* have the expected vertex order:

```
msp.add_3dface([(0, 0), (10, 0), (10, 10), (0, 10)])
```



Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	`'3DFACE'`
Factory function	*ezdxf.layouts.BaseLayout.add_3dface()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Face3d**

The class name is *Face3d* because 3dface is not a valid Python class name.

dxf.**vtx0**

Location of 1. vertex (3D Point in *WCS*)

dxf.**vtx1**

Location of 2. vertex (3D Point in *WCS*)

dxf.**vtx2**

Location of 3. vertex (3D Point in *WCS*)

dxf.**vtx3**

Location of 4. vertex (3D Point in *WCS*)

dxf.**invisible_edge**

invisible edge flag (int, default=0)

1	first edge is invisible
2	second edge is invisible
4	third edge is invisible
8	fourth edge is invisible

Combine values by adding them, e.g. 1+4 = first and third edge is invisible.

transform (*m:* Matrix44) → Face3d

Transform the 3DFACE entity by transformation matrix *m* inplace.

wcs_vertices (*close: bool = False*) → list[*Vec3*]

Returns WCS vertices, if argument *close* is True, the first vertex is also returned as closing last vertex.

Returns 4 vertices when *close* is False and 5 vertices when *close* is True. Some edges may have zero-length. This is a compatibility interface to SOLID and TRACE. The 3DFACE entity is already defined by WCS vertices.

Solid3d

3DSOLID entity (DXF Reference) created by an ACIS geometry kernel provided by the Spatial Corp.

See also:

Ezdxf has only very limited support for ACIS based entities, for more information see the FAQ: *How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?*

Subclass of	*ezdxf.entities.Body*
DXF type	'3DSOLID'
Factory function	*ezdxf.layouts.BaseLayout.add_3dsolid()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Solid3d**

Same attributes and methods as parent class *Body*.

dxf.**history_handle**

Handle to history object.

Arc

The ARC entity (DXF Reference) represents a circular arc, which is defined by the DXF attributes `dxf.center`, `dxf.radius`, `dxf.start_angle` and `dxf.end_angle`. The arc-curve goes always from `dxf.start_angle` to `dxf.end_angle` in counter-clockwise orientation around the `dxf.extrusion` vector, which is (0, 0, 1) by default and the usual case for 2D arcs. The ARC entity has *OCS* coordinates.

The helper tool *ezdxf.math.ConstructionArc* supports creating arcs from various scenarios, like from 3 points or 2 points and an angle or 2 points and a radius and the *upright* module can convert inverted extrusion vectors from (0, 0, -1) to (0, 0, 1) without changing the curve.

See also:

- *Tutorial for Simple DXF Entities*, section *Arc*
- *ezdxf.math.ConstructionArc*
- *Object Coordinate System (OCS)*
- *ezdxf.upright* module

Subclass of	*ezdxf.entities.Circle*
DXF type	'ARC'
Factory function	*ezdxf.layouts.BaseLayout.add_arc()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Arc**

dxf.**center**

Center point of arc (2D/3D Point in *OCS*)

dxf.**radius**

Radius of arc (float)

dxf.**start_angle**

Start angle in degrees (float)

dxf.**end_angle**

End angle in degrees (float)

start_point

Returns the start point of the arc in *WCS*, takes the *OCS* into account.

end_point

Returns the end point of the arc in *WCS*, takes the *OCS* into account.

angles(*num: int*) → Iterator[float]

Yields *num* angles from start- to end angle in degrees in counter-clockwise orientation. All angles are normalized in the range from [0, 360).

flattening(*sagitta: float*) → Iterator[*Vec3*]

Approximate the arc by vertices in *WCS*, the argument sagitta defines the maximum distance from the center of an arc segment to the center of its chord.

transform(*m:* Matrix44) → Arc

Transform ARC entity by transformation matrix *m* inplace. Raises `NonUniformScalingError()` for non-uniform scaling.

to_ellipse(*replace=True*) → *Ellipse*

Convert the CIRCLE/ARC entity to an *`Ellipse`* entity.

Adds the new ELLIPSE entity to the entity database and to the same layout as the source entity.

Parameters
replace – replace (delete) source entity by ELLIPSE entity if `True`

to_spline(*replace=True*) → *Spline*

Convert the CIRCLE/ARC entity to a *`Spline`* entity.

Adds the new SPLINE entity to the entity database and to the same layout as the source entity.

Parameters
replace – replace (delete) source entity by SPLINE entity if `True`

construction_tool() → *ConstructionArc*

Returns the 2D construction tool *`ezdxf.math.ConstructionArc`* but the extrusion vector is ignored.

apply_construction_tool(*arc:* ConstructionArc) → Arc

Set ARC data from the construction tool *`ezdxf.math.ConstructionArc`* but the extrusion vector is ignored.

Body

BODY entity (DXF Reference) created by an ACIS geometry kernel provided by the Spatial Corp.

See also:

Ezdxf has only very limited support for ACIS based entities, for more information see the FAQ: *How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?*

Subclass of	*`ezdxf.entities.DXFGraphic`*
DXF type	`'BODY'`
Factory function	*`ezdxf.layouts.BaseLayout.add_body()`*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Body**

dxf.**version**
Modeler format version number, default value is 1

dxf.**flags**
Require DXF R2013.

dxf.**uid**
Require DXF R2013.

property acis_data: bytes | Sequence[str]
Returns *SAT* data for DXF R2000 up to R2010 and *SAB* data for DXF R2013 and later

property sat: Sequence[str]
Get/Set *SAT* data as sequence of strings.

property sab: bytes
Get/Set *SAB* data as bytes.

property has_binary_data
Returns `True` if the entity contains *SAB* data and `False` if the entity contains *SAT* data.

tostring() → str
Returns ACIS *SAT* data as a single string if the entity has SAT data.

Circle

The CIRCLE entity (DXF Reference) defined by the DXF attributes `dxf.center` and `dxf.radius`. The CIRCLE entity has *OCS* coordinates.

See also:

- *Tutorial for Simple DXF Entities*, section *Circle*
- *ezdxf.math.ConstructionCircle*
- *Object Coordinate System (OCS)*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'CIRCLE'
Factory function	*ezdxf.layouts.BaseLayout.add_circle()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Circle**

dxf.**center**
Center point of circle (2D/3D Point in *OCS*)

dxf.**radius**
Radius of circle (float)

vertices (*angles: Iterable[float]*) → Iterator[*Vec3*]

Yields the vertices of the circle of all given *angles* as `Vec3` instances in *WCS*.

Parameters
angles – iterable of angles in *OCS* as degrees, angle goes counter-clockwise around the extrusion vector, and the OCS x-axis defines 0-degree.

flattening (*sagitta: float*) → Iterator[*Vec3*]

Approximate the circle by vertices in *WCS* as `Vec3` instances. The argument sagitta is the maximum distance from the center of an arc segment to the center of its chord. Yields a closed polygon where the start vertex is equal to the end vertex!

transform (*m:* Matrix44) → Circle

Transform the CIRCLE entity by transformation matrix *m* inplace. Raises `NonUniformScalingError()` for non-uniform scaling.

translate (*dx: float*, *dy: float*, *dz: float*) → Circle

Optimized CIRCLE/ARC translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).

to_ellipse (*replace=True*) → *Ellipse*

Convert the CIRCLE/ARC entity to an `Ellipse` entity.

Adds the new ELLIPSE entity to the entity database and to the same layout as the source entity.

Parameters
replace – replace (delete) source entity by ELLIPSE entity if `True`

to_spline (*replace=True*) → *Spline*

Convert the CIRCLE/ARC entity to a `Spline` entity.

Adds the new SPLINE entity to the entity database and to the same layout as the source entity.

Parameters
replace – replace (delete) source entity by SPLINE entity if `True`

Dimension

The DIMENSION entity (DXF Reference) represents several types of dimensions in many orientations and alignments. The basic types of dimensioning are linear, radial, angular, ordinate, and arc length.

For more information about dimensions see the online help from AutoDesk: About the Types of Dimensions

---

Important: The DIMENSION entity is reused to create dimensional constraints, such entities do not have an associated geometrical block nor a dimension type group code (2) and reside on layer `*ADSK_CONSTRAINTS`. Use property `Dimension.is_dimensional_constraint` to check for this objects. Dimensional constraints are not documented in the DXF reference and not supported by *ezdxf*.

See also:

- *Tutorial for Linear Dimensions*
- *Tutorial for Radius Dimensions*
- *Tutorial for Diameter Dimensions*
- *Tutorial for Angular Dimensions*
- *Tutorial for Ordinate Dimensions*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'DIMENSION'
factory function	see table below
Inherited DXF attributes	*Common graphical DXF attributes*

Factory Functions

Linear and Rotated Dimension (DXF)	*add_linear_dim()*
Aligned Dimension (DXF)	*add_aligned_dim()*
Angular Dimension (DXF)	*add_angular_dim_2l()*
Angular 3P Dimension (DXF)	*add_angular_dim_3p()*
Angular Dimension by center, radius, angles	*add_angular_dim_cra()*
Angular Dimension by ConstructionArc	*add_angular_dim_arc()*
Diameter Dimension (DXF)	*add_diameter_dim()*
Radius Dimension (DXF)	*add_radius_dim()*
Ordinate Dimension (DXF)	*add_ordinate_dim()*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Dimension**

There is only one *Dimension* class to represent all different dimension types.

dxf.**version**

Version number: 0 = R2010. (int, DXF R2010)

dxf.**geometry**

Name of the BLOCK that contains the entities that make up the dimension picture.

For AutoCAD this graphical representation is mandatory, otherwise AutoCAD will not open the DXF document. BricsCAD will render the DIMENSION entity by itself, if the graphical representation is not present, but displays the BLOCK content if present.

dxf.**dimstyle**

Dimension style (*DimStyle*) name as string.

dxf.**dimtype**

Values 0-6 are integer values that represent the dimension type. Values 32, 64, and 128 are bit values, which are added to the integer values.

0	Linear and Rotated Dimension (DXF)
1	Aligned Dimension (DXF)
2	Angular Dimension (DXF)
3	Diameter Dimension (DXF)
4	Radius Dimension (DXF)
5	Angular 3P Dimension (DXF)
6	Ordinate Dimension (DXF)
8	subclass *ezdxf.entities.ArcDimension* introduced in DXF R2004
32	Indicates that graphical representation `geometry` is referenced by this dimension only. (always set in DXF R13 and later)
64	Ordinate type. This is a bit value (bit 7) used only with integer value 6. If set, ordinate is *X-type*; if not set, ordinate is *Y-type*
128	This is a bit value (bit 8) added to the other `dimtype` values if the dimension text has been positioned at a user-defined location rather than at the default location

`dxf.`**`defpoint`**

Definition point for all dimension types. (3D Point in *WCS*)

- Linear- and rotated dimension: `dxf.defpoint` specifies the dimension line location.
- Arc- and angular dimension: `dxf.defpoint` and `dxfdefpoint4` specify the endpoints of the line used to determine the second extension line.

`dxf.`**`defpoint2`**

Definition point for linear- and angular dimensions. (3D Point in *WCS*)

- Linear- and rotated dimension: The `dxf.defpoint2` specifies the start point of the first extension line.
- Arc- and angular dimension: The `dxf.defpoint2` and `dxf.defpoint3` specify the endpoints of the line used to determine the first extension line.

`dxf.`**`defpoint3`**

Definition point for linear- and angular dimensions. (3D Point in *WCS*)

- Linear- and rotated dimension: The `dxf.defpoint3` specifies the start point of the second extension line.
- Arc- and angular dimension: The `dxf.defpoint2` and `dxf.defpoint3` specify the endpoints of the line used to determine the first extension line.

`dxf.`**`defpoint4`**

Definition point for diameter-, radius-, and angular dimensions. (3D Point in *WCS*)

The `dxf.defpoint` and `dxf.defpoint4` specify the endpoints of the line used to determine the second extension line for arc- and angular dimension:

`dxf.`**`defpoint5`**

This point defines the location of the arc for angular dimensions. (3D Point in *OCS*)

`dxf.`**`angle`**

Rotation angle of linear and rotated dimensions in degrees. (float)

`dxf.`**`leader_length`**

Leader length for radius and diameter dimensions. (float)

`dxf.text_midpoint`

Middle point of dimension text. (3D Point in *OCS*)

`dxf.insert`

Insertion point for clones of a linear dimensions. (3D Point in *OCS*)

This value translates the content of the associated anonymous block for cloned linear dimensions, similar to the `insert` attribute of the *`Insert`* entity.

`dxf.attachment_point`

Text attachment point (int, DXF R2000), default value is 5.

1	Top left
2	Top center
3	Top right
4	Middle left
5	Middle center
6	Middle right
7	Bottom left
8	Bottom center
9	Bottom right

`dxf.line_spacing_style`

Dimension text line-spacing style (int, DXF R2000), default value is 1.

1	At least (taller characters will override)
2	Exact (taller characters will not override)

`dxf.line_spacing_factor`

Dimension text-line spacing factor. (float, DXF R2000)

Percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00.

`dxf.actual_measurement`

Actual measurement (float, DXF R2000), this is an optional attribute and often not present. (read-only value)

`dxf.text`

Dimension text explicitly entered by the user (str), default value is an empty string.

If empty string or "<>", the dimension measurement is drawn as the text, if " " (one blank space), the text is suppressed. Anything else will be displayed as the dimension text.

`dxf.oblique_angle`

The optional `dxf.oblique_angle` defines the angle of the extension lines for linear dimension.

`dxf.text_rotation`

Defines is the rotation angle of the dimension text away from its default orientation (the direction of the dimension line). (float)

`dxf.horizontal_direction`

Indicates the horizontal direction for the dimension entity (float).

This attribute determines the orientation of dimension text and lines for horizontal, vertical, and rotated linear dimensions. This value is the negative of the angle in the OCS xy-plane between the dimension line and the OCS x-axis.

property dimtype: int

dxf.dimtype without binary flags (32, 62, 128).

property is_dimensional_constraint: bool

Returns `True` if the DIMENSION entity is a dimensional constraint object.

get_dim_style() → *DimStyle*

Returns the associated *DimStyle* entity.

get_geometry_block() → *BlockLayout* | None

Returns *BlockLayout* of associated anonymous dimension block, which contains the entities that make up the dimension picture. Returns `None` if block name is not set or the BLOCK itself does not exist

get_measurement() → float | *Vec3*

Returns the actual dimension measurement in *WCS* units, no scaling applied for linear dimensions. Returns angle in degrees for angular dimension from 2 lines and angular dimension from 3 points. Returns vector from origin to feature location for ordinate dimensions.

override() → DimStyleOverride

Returns the *DimStyleOverride* object.

render() → None

Renders the graphical representation of the DIMENSION entity as DXF primitives (TEXT, LINE, ARC, …) into an anonymous content BLOCK.

transform(*m:* Matrix44) → Dimension

Transform the DIMENSION entity by transformation matrix *m* inplace.

Raises `NonUniformScalingError()` for non uniform scaling.

virtual_entities() → Iterator[DXFGraphic]

Yields the graphical representation of the anonymous content BLOCK as virtual DXF primitives (LINE, ARC, TEXT, …).

These virtual entities are located at the original location of the DIMENSION entity, but they are not stored in the entity database, have no handle and are not assigned to any layout.

explode(*target_layout:* BaseLayout | *None* = *None*) → *EntityQuery*

Explodes the graphical representation of the DIMENSION entity as DXF primitives (LINE, ARC, TEXT, …) into the target layout, `None` for the same layout as the source DIMENSION entity.

Returns an *EntityQuery* container containing all DXF primitives.

Parameters

target_layout – target layout for the DXF primitives, `None` for same layout as source DIMENSION entity.

DimStyleOverride

All of the *DimStyle* attributes can be overridden for each *Dimension* entity individually.

The *DimStyleOverride* class manages all the complex dependencies between *DimStyle* and *Dimension*, the different features of all DXF versions and the rendering process to create the *Dimension* picture as BLOCK, which is required for AutoCAD.

class ezdxf.entities.**DimStyleOverride**

dimension

Base *Dimension* entity.

dimstyle

By *dimension* referenced *DimStyle* entity.

dimstyle_attribs

Contains all overridden attributes of *dimension*, as a `dict` with *DimStyle* attribute names as keys.

__getitem__(*key: str*) → Any

Returns DIMSTYLE attribute *key*, see also *get()*.

__setitem__(*key: str*, *value: Any*) → None

Set DIMSTYLE attribute *key* in *dimstyle_attribs*.

__delitem__(*key: str*) → None

Deletes DIMSTYLE attribute *key* from *dimstyle_attribs*, ignores `KeyErrors` silently.

get(*attribute: str*, *default: Any = None*) → Any

Returns DIMSTYLE *attribute* from override dict *dimstyle_attribs* or base *DimStyle*.

Returns *default* value for attributes not supported by DXF R12. This is a hack to use the same algorithm to render DXF R2000 and DXF R12 DIMENSION entities. But the DXF R2000 attributes are not stored in the DXF R12 file! This method does not catch invalid attribute names! Check debug log for ignored DIMSTYLE attributes.

pop(*attribute: str*, *default: Any = None*) → Any

Returns DIMSTYLE *attribute* from override dict *dimstyle_attribs* and removes this *attribute* from override dict.

update(*attribs: dict*) → None

Update override dict *dimstyle_attribs*.

Parameters
attribs – `dict` of DIMSTYLE attributes

commit() → None

Writes overridden DIMSTYLE attributes into ACAD:DSTYLE section of XDATA of the DIMENSION entity.

get_arrow_names() → tuple[str, str]

Get arrow names as strings like 'ARCHTICK' as tuple (dimblk1, dimblk2).

set_arrows(*blk: str | None = None*, *blk1: str | None = None*, *blk2: str | None = None*, *ldrblk: str | None = None*, *size: float | None = None*) → None

Set arrows or user defined blocks and disable oblique stroke as tick.

Parameters

- **blk** – defines both arrows at once as name str or user defined block
- **blk1** – defines left arrow as name str or as user defined block
- **blk2** – defines right arrow as name str or as user defined block
- **ldrblk** – defines leader arrow as name str or as user defined block
- **size** – arrow size in drawing units

set_tick(*size: float = 1*) → None

Use oblique stroke as tick, disables arrows.

Parameters
size – arrow size in daring units

set_text_align(*halign: str | None = None*, *valign: str | None = None*, *vshift: float | None = None*) → None

Set measurement text alignment, *halign* defines the horizontal alignment, *valign* defines the vertical alignment, *above1* and *above2* means above extension line 1 or 2 and aligned with extension line.

Parameters

- **halign** – *left*, *right*, *center*, *above1*, *above2*, requires DXF R2000+
- **valign** – *above*, *center*, *below*
- **vshift** – vertical text shift, if *valign* is *center*; >0 shift upward, <0 shift downwards

set_tolerance(*upper: float*, *lower: float | None = None*, *hfactor: float | None = None*, *align:* MTextLineAlignment *| None = None*, *dec: int | None = None*, *leading_zeros: bool | None = None*, *trailing_zeros: bool | None = None*) → None

Set tolerance text format, upper and lower value, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

- **upper** – upper tolerance value
- **lower** – lower tolerance value, if None same as upper
- **hfactor** – tolerance text height factor in relation to the dimension text height
- **align** – tolerance text alignment enum *ezdxf.enums.MTextLineAlignment*
- **dec** – Sets the number of decimal places displayed
- **leading_zeros** – suppress leading zeros for decimal dimensions if `False`
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`

set_limits(*upper: float*, *lower: float*, *hfactor: float | None = None*, *dec: int | None = None*, *leading_zeros: bool | None = None*, *trailing_zeros: bool | None = None*) → None

Set limits text format, upper and lower limit values, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

- **upper** – upper limit value added to measurement value
- **lower** – lower limit value subtracted from measurement value
- **hfactor** – limit text height factor in relation to the dimension text height
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if `False`, requires DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`, requires DXF R2000+

set_text_format(*prefix: str = ''*, *postfix: str = ''*, *rnd: float | None = None*, *dec: int | None = None*, *sep: str | None = None*, *leading_zeros: bool | None = None*, *trailing_zeros: bool | None = None*) → None

Set dimension text format, like prefix and postfix string, rounding rule and number of decimal places.

Parameters

- **prefix** – dimension text prefix text as string
- **postfix** – dimension text postfix text as string
- **rnd** – Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.
- **dec** – Sets the number of decimal places displayed for the primary units of a dimension. requires DXF R2000+
- **sep** – "." or "," as decimal separator
- **leading_zeros** – suppress leading zeros for decimal dimensions if `False`
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`

set_dimline_format (*color: int | None = None*, *linetype: str | None = None*, *lineweight: int | None = None*, *extension: float | None = None*, *disable1: bool | None = None*, *disable2: bool | None = None*)

Set dimension line properties.

Parameters

- **color** – color index
- **linetype** – linetype as string
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length
- **disable1** – True to suppress first part of dimension line
- **disable2** – True to suppress second part of dimension line

set_extline_format (*color: int | None = None*, *lineweight: int | None = None*, *extension: float | None = None*, *offset: float | None = None*, *fixed_length: float | None = None*)

Set common extension line attributes.

Parameters

- **color** – color index
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length above dimension line
- **offset** – offset from measurement point
- **fixed_length** – set fixed length extension line, length below the dimension line

set_extline1 (*linetype: str | None = None*, *disable=False*)

Set attributes of the first extension line.

Parameters

- **linetype** – linetype for the first extension line
- **disable** – disable first extension line if `True`

set_extline2 (*linetype: str | None = None*, *disable=False*)

Set attributes of the second extension line.

Parameters

- **linetype** – linetype for the second extension line
- **disable** – disable the second extension line if `True`

set_text (*text: str = '<>'*) → None

Set dimension text.

- *text* = " " to suppress dimension text
- *text* = "" or "<>" to use measured distance as dimension text
- otherwise display *text* literally

shift_text (*dh: float*, *dv: float*) → None

Set relative text movement, implemented as user location override without leader.

Parameters

- **dh** – shift text in text direction
- **dv** – shift text perpendicular to text direction

set_location (*location:* UVec, *leader=False*, *relative=False*) → None

Set text location by user, special version for linear dimensions, behaves for other dimension types like *user_location_override()*.

Parameters

- **location** – user defined text location
- **leader** – create leader from text to dimension line
- **relative** – *location* is relative to default location.

user_location_override (*location:* UVec) → None

Set text location by user, *location* is relative to the origin of the UCS defined in the *render()* method or WCS if the *ucs* argument is `None`.

render (*ucs:* UCS | *None = None*, *discard=False*) → BaseDimensionRenderer

Starts the dimension line rendering process and also writes overridden dimension style attributes into the DSTYLE XDATA section. The rendering process "draws" the graphical representation of the DIMENSION entity as DXF primitives (TEXT, LINE, ARC, …) into an anonymous content BLOCK.

You can discard the content BLOCK for a friendly CAD applications like BricsCAD, because the rendering of the dimension entity is done automatically by BricsCAD if the content BLOCK is missing, and the result is in most cases better than the rendering done by *ezdxf*.

AutoCAD does not render DIMENSION entities automatically, therefore I see AutoCAD as an unfriendly CAD application.

Parameters

- **ucs** – user coordinate system
- **discard** – discard the content BLOCK created by *ezdxf*, this works for BricsCAD, AutoCAD refuses to open DXF files containing DIMENSION entities without a content BLOCK

Returns

The rendering object of the DIMENSION entity for analytics

ArcDimension

The ARC_DIMENSION entity was introduced in DXF R2004 and is **not** documented in the DXF reference.

See also:

Tutorial for Arc Dimensions

Subclass of	*ezdxf.entities.Dimension*
DXF type	'ARC_DIMENSION'
factory function	• *add_arc_dim_3p()* • *add_arc_dim_cra()* • *add_arc_dim_arc()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	R2004 / AC1018

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**ArcDimension**

dxf.**defpoint2**
: start point of first extension line in *OCS*

dxf.**defpoint3**
: start point of second extension line in *OCS*

dxf.**defpoint4**
: center point of arc in *OCS*

dxf.**start_angle**

dxf.**end_angle**

dxf.**is_partial**

dxf.**has_leader**

dxf.**leader_point1**

dxf.**leader_point2**

dimtype
: Returns always 8.

Ellipse

The ELLIPSE entity (DXF Reference) is an elliptic 3D curve defined by the DXF attributes `dxf.center`, the `dxf.major_axis` vector and the `dxf.extrusion` vector.

The `dxf.ratio` attribute is the ratio of minor axis to major axis and has to be smaller or equal 1. The `dxf.start_param` and `dxf.end_param` attributes defines the starting- and the end point of the ellipse, a full ellipse goes from 0 to 2π. The curve always goes from start- to end param in counter clockwise orientation.

The `dxf.extrusion` vector defines the normal vector of the ellipse plane. The minor axis direction is calculated by `dxf.extrusion` cross `dxf.major_axis`. The default extrusion vector (0, 0, 1) defines an ellipse plane parallel to xy-plane of the *WCS*.

All coordinates and vectors in *WCS*.

See also:

- *Tutorial for Simple DXF Entities*, section *Ellipse*
- *ezdxf.math.ConstructionEllipse*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	`'ELLIPSE'`
factory function	*add_ellipse()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

class ezdxf.entities.**Ellipse**

dxf.**center**
: Center point of circle (2D/3D Point in *WCS*)

dxf.**major_axis**
: Endpoint of major axis, relative to the `dxf.center` (Vec3), default value is (1, 0, 0).

dxf.**ratio**
: Ratio of minor axis to major axis (float), has to be in range from 0.000001 to 1.0, default value is 1.

dxf.**start_param**
: Start parameter (float), default value is 0.

dxf.**end_param**
: End parameter (float), default value is 2π.

start_point
: Returns the start point of the ellipse in WCS.

end_point
: Returns the end point of the ellipse in WCS.

minor_axis
: Returns the minor axis of the ellipse as `Vec3` in WCS.

construction_tool() → *ConstructionEllipse*
: Returns construction tool *ezdxf.math.ConstructionEllipse*.

apply_construction_tool(*e:* ConstructionEllipse) → Ellipse
: Set ELLIPSE data from construction tool *ezdxf.math.ConstructionEllipse*.

vertices (*params: Iterable[float]*) → Iterable[*Vec3*]

Yields vertices on ellipse for iterable *params* in WCS.

Parameters

params – param values in the range from 0 to 2π in radians, param goes counter-clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

flattening (*distance: float*, *segments: int = 8*) → Iterable[*Vec3*]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided. Returns a closed polygon for a full ellipse where the start vertex has the same value as the end vertex.

Parameters

- **distance** – maximum distance from the projected curve point onto the segment chord.
- **segments** – minimum segment count

params (*num: int*) → Iterable[float]

Returns *num* params from start- to end param in counter-clockwise order.

All params are normalized in the range [0, 2π).

transform (*m:* Matrix44) → Ellipse

Transform the ELLIPSE entity by transformation matrix *m* inplace.

translate (*dx: float*, *dy: float*, *dz: float*) → Ellipse

Optimized ELLIPSE translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).

to_spline (*replace=True*) → *Spline*

Convert ELLIPSE to a `Spline` entity.

Adds the new SPLINE entity to the entity database and to the same layout as the source entity.

Parameters

replace – replace (delete) source entity by SPLINE entity if `True`

classmethod from_arc (*entity: DXFGraphic*) → Ellipse

Create a new virtual ELLIPSE entity from an ARC or a CIRCLE entity.

The new SPLINE entity has no owner, no handle, is not stored in the entity database nor assigned to any layout!

Hatch

The HATCH entity (DXF Reference) fills a closed area defined by one or more boundary paths by a hatch pattern, a solid fill, or a gradient fill.

All points in *OCS* as (x, y) tuples (`Hatch.dxf.elevation` is the z-axis value).

There are two different hatch pattern default scaling, depending on the HEADER variable $MEASUREMENT, one for ISO measurement (m, cm, mm, …) and one for imperial measurement (in, ft, yd, …).

The default scaling for predefined hatch pattern will be chosen according this measurement setting in the HEADER section, this replicates the behavior of BricsCAD and other CAD applications. *Ezdxf* uses the ISO pattern definitions as a base line and scales this pattern down by factor 1/25.6 for imperial measurement usage. The pattern scaling is independent from the drawing units of the document defined by the HEADER variable $INSUNITS.

See also:

Tutorial for Hatch and *DXF Units*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'HATCH'
Factory function	*ezdxf.layouts.BaseLayout.add_hatch()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

Boundary paths classes

Path manager: *BoundaryPaths*

- *PolylinePath*
- ***EdgePath***
  - *LineEdge*
  - *ArcEdge*
  - *EllipseEdge*
  - *SplineEdge*

Pattern and gradient classes

- *Pattern*
- *PatternLine*
- Gradien

class ezdxf.entities.**Hatch**

dxf.**pattern_name**

Pattern name as string

dxf.**solid_fill**

1	solid fill, use method *Hatch.set_solid_fill()*
0	pattern fill, use method *Hatch.set_pattern_fill()*

dxf.**associative**

1	associative hatch
0	not associative hatch

Associations are not managed by *ezdxf*.

dxf.**hatch_style**

0	normal
1	outer
2	ignore

(search AutoCAD help for more information)

dxf.**pattern_type**

0	user
1	predefined
2	custom

dxf.**pattern_angle**
The actual pattern rotation angle in degrees (float). Changing this value does not rotate the pattern, use *set_pattern_angle()* for this task.

dxf.**pattern_scale**
The actual pattern scale factor (float). Changing this value does not scale the pattern use *set_pattern_scale()* for this task.

dxf.**pattern_double**
1 = double pattern size else 0. (int)

dxf.**n_seed_points**
Count of seed points (use `get_seed_points()`)

dxf.**elevation**
Z value represents the elevation height of the *OCS*. (float)

paths
BoundaryPaths object.

pattern
Pattern object.

gradient
Gradient object.

seeds
A list of seed points as (x, y) tuples.

property has_solid_fill: bool
`True` if entity has a solid fill. (read only)

property has_pattern_fill: bool
`True` if entity has a pattern fill. (read only)

property has_gradient_data: bool
`True` if entity has a gradient fill. A hatch with gradient fill has also a solid fill. (read only)

property bgcolor: Tuple[int, int, int] | None

Set pattern fill background color as (r, g, b)-tuple, rgb values in the range [0, 255] (read/write/del)

usage:

```
r, g, b = entity.bgcolor  # get pattern fill background color
entity.bgcolor = (10, 20, 30)  # set pattern fill background color
del entity.bgcolor  # delete pattern fill background color
```

set_pattern_definition(*lines: Sequence*, *factor: float = 1*, *angle: float = 0*) → None

Setup pattern definition by a list of definition lines and the definition line is a 4-tuple (angle, base_point, offset, dash_length_items). The pattern definition should be designed for a pattern scale factor of 1 and a pattern rotation angle of 0.

- angle: line angle in degrees
- base-point: (x, y) tuple
- offset: (dx, dy) tuple
- dash_length_items: list of dash items (item > 0 is a line, item < 0 is a gap and item == 0.0 is a point)

Parameters

- **lines** – list of definition lines
- **factor** – pattern scale factor
- **angle** – rotation angle in degrees

set_pattern_scale(*scale: float*) → None

Sets the pattern scale factor and scales the pattern definition.

The method always starts from the original base scale, the `set_pattern_scale(1)` call resets the pattern scale to the original appearance as defined by the pattern designer, but only if the pattern attribute *dxf.pattern_scale* represents the actual scale, it cannot restore the original pattern scale from the pattern definition itself.

Parameters

scale – pattern scale factor

set_pattern_angle(*angle: float*) → None

Sets the pattern rotation angle and rotates the pattern definition.

The method always starts from the original base rotation of 0, the `set_pattern_angle(0)` call resets the pattern rotation angle to the original appearance as defined by the pattern designer, but only if the pattern attribute *dxf.pattern_angle* represents the actual pattern rotation, it cannot restore the original rotation angle from the pattern definition itself.

Parameters

angle – pattern rotation angle in degrees

set_solid_fill(*color: int = 7*, *style: int = 1*, *rgb: RGB | None = None*)

Set the solid fill mode and removes all gradient and pattern fill related data.

Parameters

- **color** – *AutoCAD Color Index (ACI)*, (0 = BYBLOCK; 256 = BYLAYER)
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)

- **rgb** – true color value as (r, g, b)-tuple - has higher priority than *color*. True color support requires DXF R2000.

set_pattern_fill (*name: str*, *color: int = 7*, *angle: float = 0.0*, *scale: float = 1.0*, *double: int = 0*, *style: int = 1*, *pattern_type: int = 1*, *definition=None*) → None

Sets the pattern fill mode and removes all gradient related data.

The pattern definition should be designed for a scale factor 1 and a rotation angle of 0 degrees. The predefined hatch pattern like "ANSI33" are scaled according to the HEADER variable $MEASUREMENT for ISO measurement (m, cm, …), or imperial units (in, ft, …), this replicates the behavior of BricsCAD.

Parameters

- **name** – pattern name as string
- **color** – pattern color as *AutoCAD Color Index (ACI)*
- **angle** – pattern rotation angle in degrees
- **scale** – pattern scale factor
- **double** – double size flag
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **pattern_type** – pattern type (0 = user-defined; 1 = predefined; 2 = custom)
- **definition** – list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], see `set_pattern_definition()`

set_gradient (*color1: Tuple[int, int, int] = (0, 0, 0)*, *color2: Tuple[int, int, int] = (255, 255, 255)*, *rotation: float = 0.0*, *centered: float = 0.0*, *one_color: int = 0*, *tint: float = 0.0*, *name: str = 'LINEAR'*) → None

Sets the gradient fill mode and removes all pattern fill related data, requires DXF R2004 or newer. A gradient filled hatch is also a solid filled hatch.

Valid gradient type names are:

- "LINEAR"
- "CYLINDER"
- "INVCYLINDER"
- "SPHERICAL"
- "INVSPHERICAL"
- "HEMISPHERICAL"
- "INVHEMISPHERICAL"
- "CURVED"
- "INVCURVED"

Parameters

- **color1** – (r, g, b)-tuple for first color, rgb values as int in the range [0, 255]
- **color2** – (r, g, b)-tuple for second color, rgb values as int in the range [0, 255]
- **rotation** – rotation angle in degrees
- **centered** – determines whether the gradient is centered or not
- **one_color** – 1 for gradient from *color1* to tinted *color1*

- **tint** – determines the tinted target *color1* for a one color gradient. (valid range 0.0 to 1.0)
- **name** – name of gradient type, default "LINEAR"

set_seed_points(*points: Iterable[tuple[float, float]]*) → None

Set seed points, *points* is an iterable of (x, y)-tuples. I don't know why there can be more than one seed point. All points in *OCS* (`Hatch.dxf.elevation` is the Z value)

transform(*m:* Matrix44) → *Hatch*

Transform entity by transformation matrix *m* inplace.

associate(*path: AbstractBoundaryPath*, *entities: Iterable[*DXFEntity*]*)

Set association from hatch boundary *path* to DXF geometry *entities*.

A HATCH entity can be associative to a base geometry, this association is **not** maintained nor verified by *ezdxf*, so if you modify the base geometry the geometry of the boundary path is not updated and no verification is done to check if the associated geometry matches the boundary path, this opens many possibilities to create invalid DXF files: USE WITH CARE!

remove_association()

Remove associated path elements.

Boundary Paths

The hatch entity is build by different path types, these are the filter flags for the `Hatch.dxf.hatch_style`:

- EXTERNAL: defines the outer boundary of the hatch
- OUTERMOST: defines the first tier of inner hatch boundaries
- DEFAULT: default boundary path

As you will learn in the next sections, these are more the recommended usage type for the flags, but the fill algorithm doesn't care much about that, for instance an OUTERMOST path doesn't have to be inside the EXTERNAL path.

Island Detection

In general the island detection algorithm works always from outside to inside and alternates filled and unfilled areas. The area between then 1st and the 2nd boundary is filled, the area between the 2nd and the 3rd boundary is unfilled and so on. The different hatch styles defined by the `Hatch.dxf.hatch_style` attribute are created by filtering some boundary path types.

Hatch Style

- HATCH_STYLE_IGNORE: Ignores all paths except the paths marked as EXTERNAL, if there are more than one path marked as EXTERNAL, they are filled in NESTED style. Creates no hatch if no path is marked as EXTERNAL.
- HATCH_STYLE_OUTERMOST: Ignores all paths marked as DEFAULT, remaining EXTERNAL and OUTERMOST paths are filled in NESTED style. Creates no hatch if no path is marked as EXTERNAL or OUTERMOST.
- HATCH_STYLE_NESTED: Use all existing paths.

Hatch Boundary Classes

class ezdxf.entities.**BoundaryPaths**

Defines the borders of the hatch, a hatch can consist of more than one path.

paths

List of all boundary paths. Contains *PolylinePath* and *EdgePath* objects. (read/write)

external_paths() → Iterable[AbstractBoundaryPath]

Iterable of external paths, could be empty.

outermost_paths() → Iterable[AbstractBoundaryPath]

Iterable of outermost paths, could be empty.

default_paths() → Iterable[AbstractBoundaryPath]

Iterable of default paths, could be empty.

rendering_paths(*hatch_style: int = const.HATCH_STYLE_NESTED*) → Iterable[AbstractBoundaryPath]

Iterable of paths to process for rendering, filters unused boundary paths according to the given hatch style:

- NESTED: use all boundary paths
- OUTERMOST: use EXTERNAL and OUTERMOST boundary paths
- IGNORE: ignore all paths except EXTERNAL boundary paths

Yields paths in order of EXTERNAL, OUTERMOST and DEFAULT.

add_polyline_path(*path_vertices: Iterable[tuple[float, ...]]*, *is_closed: bool = True*, *flags: int = 1*) → PolylinePath

Create and add a new *PolylinePath* object.

Parameters

- **path_vertices** – iterable of polyline vertices as (x, y) or (x, y, bulge)-tuples.
- **is_closed** – 1 for a closed polyline else 0
- **flags** – external(1) or outermost(16) or default (0)

add_edge_path(*flags: int = 1*) → EdgePath

Create and add a new *EdgePath* object.

Parameters

flags – external(1) or outermost(16) or default (0)

polyline_to_edge_paths(*just_with_bulge=True*) → None

Convert polyline paths including bulge values to line- and arc edges.

Parameters

just_with_bulge – convert only polyline paths including bulge values if `True`

edge_to_polyline_paths(*distance: float*, *segments: int = 16*)

Convert all edge paths to simple polyline paths without bulges.

Parameters

- **distance** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count per curve

arc_edges_to_ellipse_edges() → None

Convert all arc edges to ellipse edges.

ellipse_edges_to_spline_edges(*num: int = 32*) → None

Convert all ellipse edges to spline edges (approximation).

Parameters

num – count of control points for a **full** ellipse, partial ellipses have proportional fewer control points but at least 3.

spline_edges_to_line_edges(*factor: int = 8*) → None

Convert all spline edges to line edges (approximation).

Parameters

factor – count of approximation segments = count of control points x factor

all_to_spline_edges(*num: int = 64*) → None

Convert all bulge, arc and ellipse edges to spline edges (approximation).

Parameters

num – count of control points for a **full** circle/ellipse, partial circles/ellipses have proportional fewer control points but at least 3.

all_to_line_edges(*num: int = 64*, *spline_factor: int = 8*) → None

Convert all bulge, arc and ellipse edges to spline edges and approximate this splines by line edges.

Parameters

- **num** – count of control points for a **full** circle/ellipse, partial circles/ellipses have proportional fewer control points but at least 3.
- **spline_factor** – count of spline approximation segments = count of control points x spline_factor

clear() → None

Remove all boundary paths.

class ezdxf.entities.**BoundaryPathType**

POLYLINE

polyline path type

EDGE

edge path type

class ezdxf.entities.**PolylinePath**

A polyline as hatch boundary path.

type

Path type as *BoundaryPathType.POLYLINE* enum

path_type_flags

(bit coded flags)

0	default
1	external
2	polyline, will be set by *ezdxf*
16	outermost

My interpretation of the *path_type_flags*, see also *Tutorial for Hatch*:

- external: path is part of the hatch outer border
- outermost: path is completely inside of one or more external paths
- default: path is completely inside of one or more outermost paths

If there are troubles with AutoCAD, maybe the hatch entity has the `Hatch.dxf.pixel_size` attribute set - delete it `del hatch.dxf.pixel_size` and maybe the problem is solved. *Ezdxf* does not use the `Hatch.dxf.pixel_size` attribute, but it can occur in DXF files created by other applications.

is_closed

`True` if polyline path is closed.

vertices

List of path vertices as (x, y, bulge)-tuples. (read/write)

source_boundary_objects

List of handles of the associated DXF entities for associative hatches. There is no support for associative hatches by *ezdxf*, you have to do it all by yourself. (read/write)

set_vertices(*vertices: Iterable[Sequence[float]]*, *is_closed: bool = True*) → None

Set new *vertices* as new polyline path, a vertex has to be a (x, y) or a (x, y, bulge)-tuple.

clear() → None

Removes all vertices and all handles to associated DXF objects (*source_boundary_objects*).

class ezdxf.entities.**EdgePath**

Boundary path build by edges. There are four different edge types: *LineEdge*, *ArcEdge*, *EllipseEdge* of *SplineEdge*. Make sure there are no gaps between edges and the edge path must be closed to be recognized as path. AutoCAD is very picky in this regard. *Ezdxf* performs no checks on gaps between the edges and does not prevent creating open loops.

Note: *ArcEdge* and *EllipseEdge* are ALWAYS represented in counter-clockwise orientation, even if an clockwise oriented edge is required to build a closed loop. To add a clockwise oriented curve swap start- and end angles and set the *ccw* flag to *False* and *ezdxf* will export a correct clockwise orientated curve.

---

type

Path type as *BoundaryPathType.EDGE* enum

path_type_flags

(bit coded flags)

0	default
1	external
16	outermost

see *PolylinePath.path_type_flags*

edges

List of boundary edges of type *LineEdge*, *ArcEdge*, *EllipseEdge* of *SplineEdge*

source_boundary_objects

Required for associative hatches, list of handles to the associated DXF entities.

clear() → None

Delete all edges.

add_line(*start:* UVec, *end:* UVec) → LineEdge

Add a *LineEdge* from *start* to *end*.

Parameters

- **start** – start point of line, (x, y)-tuple
- **end** – end point of line, (x, y)-tuple

add_arc(*center:* UVec, *radius: float = 1.0*, *start_angle: float = 0.0*, *end_angle: float = 360.0*, *ccw: bool = True*) → ArcEdge

Add an *ArcEdge*.

Adding Clockwise Oriented Arcs:

Clockwise oriented *ArcEdge* objects are sometimes necessary to build closed loops, but the *ArcEdge* objects are always represented in counter-clockwise orientation. To add a clockwise oriented *ArcEdge* you have to swap the start- and end angle and set the *ccw* flag to `False`, e.g. to add a clockwise oriented *ArcEdge* from 180 to 90 degree, add the *ArcEdge* in counter-clockwise orientation with swapped angles:

```
edge_path.add_arc(center, radius, start_angle=90, end_angle=180, ccw=False)
```

Parameters

- **center** – center point of arc, (x, y)-tuple
- **radius** – radius of circle
- **start_angle** – start angle of arc in degrees (*end_angle* for a clockwise oriented arc)
- **end_angle** – end angle of arc in degrees (*start_angle* for a clockwise oriented arc)
- **ccw** – `True` for counter-clockwise `False` for clockwise orientation

add_ellipse(*center:* UVec, *major_axis:* UVec = *(1.0, 0.0)*, *ratio: float = 1.0*, *start_angle: float = 0.0*, *end_angle: float = 360.0*, *ccw: bool = True*) → EllipseEdge

Add an *EllipseEdge*.

Adding Clockwise Oriented Ellipses:

Clockwise oriented *EllipseEdge* objects are sometimes necessary to build closed loops, but the *EllipseEdge* objects are always represented in counter-clockwise orientation. To add a clockwise oriented *EllipseEdge* you have to swap the start- and end angle and set the *ccw* flag to `False`, e.g. to add a clockwise oriented *EllipseEdge* from 180 to 90 degree, add the *EllipseEdge* in counter-clockwise orientation with swapped angles:

```
edge_path.add_ellipse(center, major_axis, ratio, start_angle=90, end_
↪angle=180, ccw=False)
```

Parameters

- **center** – center point of ellipse, (x, y)-tuple
- **major_axis** – vector of major axis as (x, y)-tuple
- **ratio** – ratio of minor axis to major axis as float
- **start_angle** – start angle of ellipse in degrees (*end_angle* for a clockwise oriented ellipse)

- **end_angle** – end angle of ellipse in degrees (*start_angle* for a clockwise oriented ellipse)
- **ccw** – `True` for counter-clockwise `False` for clockwise orientation

add_spline (*fit_points: Iterable[UVec] | None = None*, *control_points: Iterable[UVec] | None = None*, *knot_values: Iterable[float] | None = None*, *weights: Iterable[float] | None = None*, *degree: int = 3*, *periodic: int = 0*, *start_tangent:* UVec | *None = None*, *end_tangent:* UVec | *None = None*) → SplineEdge

Add a *SplineEdge*.

Parameters

- **fit_points** – points through which the spline must go, at least 3 fit points are required. list of (x, y)-tuples
- **control_points** – affects the shape of the spline, mandatory and AutoCAD crashes on invalid data. list of (x, y)-tuples
- **knot_values** – (knot vector) mandatory and AutoCAD crashes on invalid data. list of floats; *ezdxf* provides two tool functions to calculate valid knot values: *ezdxf.math.uniform_knot_vector()*, *ezdxf.math.open_uniform_knot_vector()* (default if `None`)
- **weights** – weight of control point, not mandatory, list of floats.
- **degree** – degree of spline (int)
- **periodic** – 1 for periodic spline, 0 for none periodic spline
- **start_tangent** – start_tangent as 2d vector, optional
- **end_tangent** – end_tangent as 2d vector, optional

Warning: Unlike for the spline entity AutoCAD does not calculate the necessary *knot_values* for the spline edge itself. On the contrary, if the *knot_values* in the spline edge are missing or invalid AutoCAD **crashes**.

class ezdxf.entities.**EdgeType**

LINE

ARC

ELLIPSE

SPLINE

class ezdxf.entities.**LineEdge**

Straight boundary edge.

type

Edge type as *EdgeType.LINE* enum

start

Start point as (x, y)-tuple. (read/write)

end

End point as (x, y)-tuple. (read/write)

class ezdxf.entities.**ArcEdge**

Arc as boundary edge in counter-clockwise orientation, see *EdgePath.add_arc()*.

type

Edge type as *EdgeType.ARC* enum

center

Center point of arc as (x, y)-tuple. (read/write)

radius

Arc radius as float. (read/write)

start_angle

Arc start angle in counter-clockwise orientation in degrees. (read/write)

end_angle

Arc end angle in counter-clockwise orientation in degrees. (read/write)

ccw

`True` for counter clockwise arc else `False`. (read/write)

class ezdxf.entities.**EllipseEdge**

Elliptic arc as boundary edge in counter-clockwise orientation, see *EdgePath.add_ellipse()*.

type

Edge type as *EdgeType.ELLIPSE* enum

major_axis_vector

Ellipse major axis vector as (x, y)-tuple. (read/write)

minor_axis_length

Ellipse minor axis length as float. (read/write)

radius

Ellipse radius as float. (read/write)

start_angle

Ellipse start angle in counter-clockwise orientation in degrees. (read/write)

end_angle

Ellipse end angle in counter-clockwise orientation in degrees. (read/write)

ccw

`True` for counter clockwise ellipse else `False`. (read/write)

class ezdxf.entities.**SplineEdge**

Spline as boundary edge.

type

Edge type as *EdgeType.SPLINE* enum

degree

Spline degree as int. (read/write)

rational

1 for rational spline else 0. (read/write)

periodic

1 for periodic spline else 0. (read/write)

knot_values

List of knot values as floats. (read/write)

control_points

List of control points as (x, y)-tuples. (read/write)

fit_points

List of fit points as (x, y)-tuples. (read/write)

weights

List of weights (of control points) as floats. (read/write)

start_tangent

Spline start tangent (vector) as (x, y)-tuple. (read/write)

end_tangent

Spline end tangent (vector) as (x, y)-tuple. (read/write)

Hatch Pattern Definition Classes

class ezdxf.entities.**Pattern**

lines

List of pattern definition lines (read/write). see *PatternLine*

add_line(*angle: float = 0*, *base_point:* UVec *= (0, 0)*, *offset:* UVec *= (0, 0)*, *dash_length_items: Iterable[float] | None = None*) → None

Create a new pattern definition line and add the line to the *Pattern.lines* attribute.

clear() → None

Delete all pattern definition lines.

scale(*factor: float = 1*, *angle: float = 0*) → None

Scale and rotate pattern.

Be careful, this changes the base pattern definition, maybe better use *Hatch.set_pattern_scale()* or *Hatch.set_pattern_angle()*.

Parameters

- **factor** – scaling factor
- **angle** – rotation angle in degrees

class ezdxf.entities.**PatternLine**

Represents a pattern definition line, use factory function *Pattern.add_line()* to create new pattern definition lines.

angle

Line angle in degrees. (read/write)

base_point

Base point as (x, y)-tuple. (read/write)

offset

Offset as (x, y)-tuple. (read/write)

dash_length_items

List of dash length items (item > 0 is line, < 0 is gap, 0.0 = dot). (read/write)

Hatch Gradient Fill Class

class ezdxf.entities.**Gradient**

color1

First rgb color as (r, g, b)-tuple, rgb values in range 0 to 255. (read/write)

color2

Second rgb color as (r, g, b)-tuple, rgb values in range 0 to 255. (read/write)

one_color

If *one_color* is 1 - the hatch is filled with a smooth transition between *color1* and a specified *tint* of *color1*. (read/write)

rotation

Gradient rotation in degrees. (read/write)

centered

Specifies a symmetrical gradient configuration. If this option is not selected, the gradient fill is shifted up and to the left, creating the illusion of a light source to the left of the object. (read/write)

tint

Specifies the tint (*color1* mixed with white) of a color to be used for a gradient fill of one color. (read/write)

See also:

Tutorial for Hatch Pattern Definition

Helix

The HELIX entity (DXF Reference).

The helix curve is represented by a cubic B-spline curve, therefore the HELIX entity is also derived from the SPLINE entity.

See also:

- Wikipedia article about the helix shape

Subclass of	*ezdxf.entities.Spline*
DXF type	'HELIX'
Factory function	*ezdxf.layouts.BaseLayout.add_helix()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**Helix**

All points in *WCS* as (x, y, z) tuples

dxf.**axis_base_point**

The base point of the helix axis (Vec3).

dxf.**start_point**

The starting point of the helix curve (Vec3). This also defines the base radius as the distance from the start point to the axis base point.

dxf.**axis_vector**

Defines the direction of the helix axis (Vec3).

dxf.**radius**

Defines the top radius of the helix (float).

dxf.**turn_height**

Defines the pitch (height if one helix turn) of the helix (float).

dxf.**turns**

The count of helix turns (float).

dxf.**handedness**

Helix orientation (int).

0	clock wise (left handed)
1	counter clockwise (right handed)

dxf.**constrain**

0	constrain turn height (pitch)
1	constrain count of turns
2	constrain total height

Image

The IMAGE entity (DXF Reference) represents a raster image, the image file itself is not embedded into the DXF file, it is always a separated file. The IMAGE entity is like a block reference, it can be used to add the image multiple times at different locations with different scale and rotation angles. Every IMAGE entity requires an image definition, see entity *ImageDef*. *Ezdxf* creates only images in the xy-plan, it's possible to place images in 3D space, therefore the *Image.dxf.u_pixel* and the *Image.dxf.v_pixel* vectors has to be set accordingly.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'IMAGE'
Factory function	*ezdxf.layouts.BaseLayout.add_image()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Image**

dxf.**insert**

Insertion point, lower left corner of the image (3D Point in *WCS*).

dxf.**u_pixel**

U-vector of a single pixel as (x, y, z) tuple. This vector points along the visual bottom of the image, starting at the insertion point.

dxf.**v_pixel**

V-vector of a single pixel as (x, y, z) tuple. This vector points along the visual left side of the image, starting at the insertion point.

dxf.**image_size**

Image size in pixels as (x, y) tuple

dxf.**image_def_handle**

Handle to the image definition entity, see *ImageDef*

dxf.**flags**

Image.SHOW_IMAGE	1	Show image
Image.SHOW_WHEN_NOT_ALIGNED	2	Show image when not aligned with screen
Image.USE_CLIPPING_BOUNDARY	4	Use clipping boundary
Image.USE_TRANSPARENCY	8	Transparency is on

dxf.**clipping**

Clipping state:

0	clipping off
1	clipping on

dxf.**brightness**

Brightness value in the range [0, 100], default is 50

dxf.**contrast**

Contrast value in the range [0, 100], default is 50

dxf.**fade**

Fade value in the range [0, 100], default is 0

dxf.**clipping_boundary_type**

1	Rectangular
2	Polygonal

dxf.**count_boundary_points**

Number of clip boundary vertices, this attribute is maintained by *ezdxf*.

dxf.**clip_mode**

0	Outside
1	Inside

requires DXF R2010 or newer

boundary_path

A list of vertices as pixel coordinates, Two vertices describe a rectangle, lower left corner is (-0.5, -0.5) and upper right corner is (ImageSizeX-0.5, ImageSizeY-0.5), more than two vertices is a polygon as clipping path. All vertices as pixel coordinates. (read/write)

image_def

Returns the associated IMAGEDEF entity, see *ImageDef*.

reset_boundary_path() → None

Reset boundary path to the default rectangle [(-0.5, -0.5), (ImageSizeX-0.5, ImageSizeY-0.5)].

set_boundary_path(*vertices: Iterable[*UVec*]*) → None

Set boundary path to *vertices*. Two vertices describe a rectangle (lower left and upper right corner), more than two vertices is a polygon as clipping path.

boundary_path_wcs() → list[*ezdxf.math._vector.Vec3*]

Returns the boundary/clipping path in WCS coordinates.

It's recommended to acquire the clipping path as *Path* object by the *make_path()* function:

```
from ezdxf.path import make_path

image = ...  # get image entity
clipping_path = make_path(image)
```

transform(*m:* Matrix44) → ImageBase

Transform IMAGE entity by transformation matrix *m* inplace.

Leader

The LEADER entity (DXF Reference) represents a pointer line, made up of one or more vertices (or spline fit points) and an arrowhead. The label or other content to which the *Leader* is attached is stored as a separate entity, and is not part of the *Leader* itself.

The LEADER entity uses parts of the styling infrastructure of the DIMENSION entity.

By default a *Leader* without any annotation is created. For creating more fancy leaders and annotations see the documentation provided by Autodesk or Demystifying DXF: LEADER and MULTILEADER implementation notes .

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'LEADER'
Factory function	*ezdxf.layouts.BaseLayout.add_leader()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**Leader**

dxf.**dimstyle**

Name of Dimstyle as string.

dxf.**has_arrowhead**

0	Disabled
1	Enabled

dxf.**path_type**

Leader path type:

0	Straight line segments
1	Spline

dxf.**annotation_type**

0	Created with text annotation
1	Created with tolerance annotation
2	Created with block reference annotation
3	Created without any annotation (default)

dxf.**hookline_direction**

Hook line direction flag:

0	Hookline (or end of tangent for a splined leader) is the opposite direction from the horizontal vector
1	Hookline (or end of tangent for a splined leader) is the same direction as horizontal vector (see `has_hook_line`)

dxf.**has_hookline**

0	No hookline
1	Has a hookline

dxf.**text_height**

Text annotation height in drawing units.

dxf.**text_width**

Text annotation width.

dxf.**block_color**

Color to use if leader's DIMCLRD = BYBLOCK

dxf.**annotation_handle**

Hard reference (handle) to associated annotation (*MText*, `Tolerance`, or *Insert* entity)

dxf.**normal_vector**

Extrusion vector? default is (0, 0, 1).

.dxf.**horizontal_direction**

Horizontal direction for leader, default is (1, 0, 0).

dxf.**leader_offset_block_ref**

Offset of last leader vertex from block reference insertion point, default is (0, 0, 0).

dxf.**leader_offset_annotation_placement**

Offset of last leader vertex from annotation placement point, default (0, 0, 0).

vertices

List of *Vec3* objects, representing the vertices of the leader (3D Point in *WCS*).

set_vertices(*vertices: Iterable[*UVec*]*)

Set vertices of the leader, vertices is an iterable of (x, y [,z]) tuples or *Vec3*.

transform(*m:* Matrix44) → *Leader*

Transform LEADER entity by transformation matrix *m* inplace.

virtual_entities() → Iterator[DXFGraphic]

Yields the DXF primitives the LEADER entity is build up as virtual entities.

These entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode(*target_layout:* BaseLayout | *None* = *None*) → *EntityQuery*

Explode parts of the LEADER entity as DXF primitives into target layout, if target layout is None, the target layout is the layout of the LEADER entity. This method destroys the source entity.

Returns an *EntityQuery* container referencing all DXF primitives.

Parameters

target_layout – target layout for the created DXF primitives, None for the same layout as the source entity.

Line

The LINE entity (DXF Reference) is a 3D line defined by the DXF attributes dxf.start and dxf.end. The LINE entity has *WCS* coordinates.

See also:

- *Tutorial for Simple DXF Entities*, section *Line*
- *ezdxf.math.ConstructionRay*
- *ezdxf.math.ConstructionLine*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'LINE'
Factory function	*ezdxf.layouts.BaseLayout.add_line()*
Inherited DXF Attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Line**

`dxf.start`
: start point of line (2D/3D Point in *WCS*)

`dxf.end`
: end point of line (2D/3D Point in *WCS*)

`dxf.thickness`
: Line thickness in 3D space in direction `extrusion`, default value is 0. This value should not be confused with the *`lineweight`* value.

`dxf.extrusion`
: extrusion vector, default value is (0, 0, 1)

`transform`(*m:* Matrix44) → Line
: Transform the LINE entity by transformation matrix *m* inplace.

`translate`(*dx: float*, *dy: float*, *dz: float*) → Line
: Optimized LINE translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

LWPolyline

The LWPOLYLINE entity (Lightweight POLYLINE, DXF Reference) is defined as a single graphic entity, which differs from the old-style *`Polyline`* entity, which is defined as a group of sub-entities. *`LWPolyline`* display faster (in AutoCAD) and consume less disk space, it is a planar element, therefore all points are located in the *OCS* as (x, y)-tuples (*`LWPolyline.dxf.elevation`* is the z-axis value).

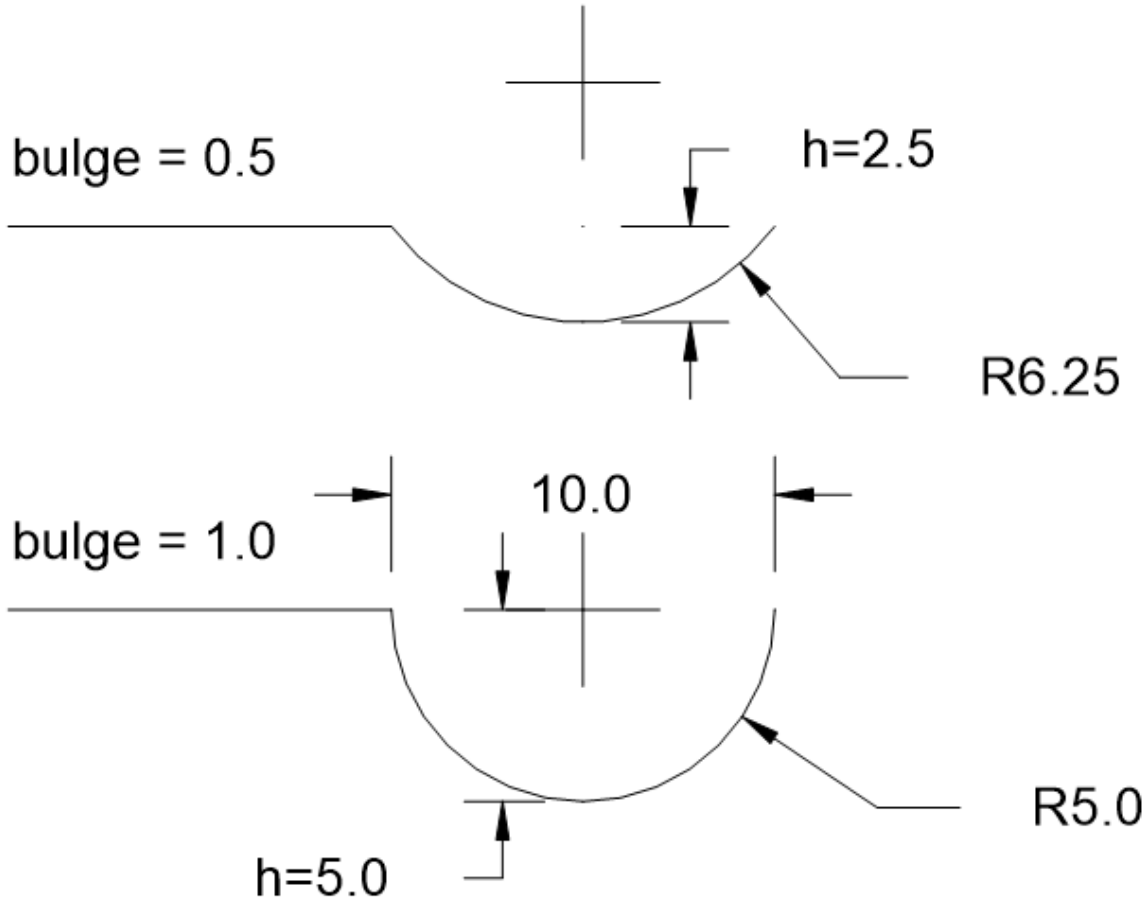
Subclass of	*`ezdxf.entities.DXFGraphic`*
DXF type	`'LWPOLYLINE'`
factory function	*`add_lwpolyline()`*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

Bulge value

The bulge value is used to create arc shaped line segments for *`Polyline`* and *`LWPolyline`* entities. The arc starts at the vertex which includes the bulge value and ends at the following vertex. The bulge value defines the ratio of the arc sagitta (versine) to half line segment length, a bulge value of 1 defines a semicircle.

The sign of the bulge value defines the side of the bulge:

- positive value (> 0): bulge is right of line (counter clockwise)
- negative value (< 0): bulge is left of line (clockwise)
- 0 = no bulge



Start- and end width

The start width and end width values defines the width in drawing units for the following line segment. To use the default width value for a line segment set value to 0.

Width and bulge values at last point

The width and bulge values of the last point has only a meaning if the polyline is closed, and they apply to the last line segment from the last to the first point.

See also:

Tutorial for LWPolyline and *Bulge Related Functions*

User Defined Point Format Codes

Code	Point Component
x	x-coordinate
y	y-coordinate
s	start width
e	end width
b	bulge value
v	(x, y [, z]) as tuple

class ezdxf.entities.**LWPolyline**

dxf.**elevation**

OCS z-axis value for all polyline points, default=0

dxf.**flags**

Constants defined in *ezdxf.lldxf.const*:

dxf.flags	Value	Description
LWPOLYLINE_CLOSED	1	polyline is closed
LWPOLYLINE_PLINEGEN	128	linetype is generated across the points

dxf.**const_width**

Constant line width (float), default value is 0.

dxf.**count**

Count of polyline points (read only), same as `len(polyline)`

property closed: bool

Get/set closed state of polyline. A closed polyline has a connection segment from the last vertex to the first vertex.

property is_closed: bool

Get closed state of LWPOLYLINE. Compatibility interface to *Polyline*

close(*state: bool = True*) → None

Set closed state of LWPOLYLINE. Compatibility interface to *Polyline*

property has_arc: bool

Returns `True` if LWPOLYLINE has an arc segment.

property has_width: bool

Returns `True` if LWPOLYLINE has any segment with width attributes or the DXF attribute const_width is not 0.

__len__() → int

Returns count of polyline points.

__getitem__(*index: int*) → Tuple[float, float, float, float, float]

Returns point at position *index* as (x, y, start_width, end_width, bulge) tuple. start_width, end_width and bulge is 0 if not present, supports extended slicing. Point format is fixed as "xyseb".

All coordinates in *OCS*.

__setitem__(*index: int*, *value: Sequence[float]*) → None

Set point at position *index* as (x, y, [start_width, [end_width, [bulge]]]) tuple. If start_width or end_width is 0 or left off the default width value is used. If the bulge value is left off, bulge is 0 by default (straight line). Does NOT support extend slicing. Point format is fixed as "xyseb".

All coordinates in *OCS*.

Parameters

- **index** – point index
- **value** – point value as (x, y, [start_width, [end_width, [bulge]]]) tuple

__delitem__(*index: int*) → None

Delete point at position *index*, supports extended slicing.

__iter__() → Iterator[Tuple[float, float, float, float, float]]

Returns iterable of tuples (x, y, start_width, end_width, bulge).

vertices() → Iterator[tuple[float, float]]

Returns iterable of all polyline points as (x, y) tuples in *OCS* (`dxf.elevation` is the z-axis value).

vertices_in_wcs() → Iterator[*Vec3*]

Returns iterable of all polyline points as Vec3(x, y, z) in *WCS*.

append(*point: Sequence[float]*, *format: str = DEFAULT_FORMAT*) → None

Append *point* to polyline, *format* specifies a user defined point format.

All coordinates in *OCS*.

Parameters

- **point** – (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, default is "xyseb", see: *format codes*

append_points(*points: Iterable[Sequence[float]]*, *format: str = DEFAULT_FORMAT*) → None

Append new *points* to polyline, *format* specifies a user defined point format.

All coordinates in *OCS*.

Parameters

- **points** – iterable of point, point is (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, default is "xyseb", see: *format codes*

insert(*pos: int*, *point: Sequence[float]*, *format: str = DEFAULT_FORMAT*) → None

Insert new point in front of positions *pos*, *format* specifies a user defined point format.

All coordinates in *OCS*.

Parameters

- **pos** – insert position
- **point** – point data
- **format** – format string, default is "xyseb", see: *format codes*

clear() → None

Remove all points.

get_points (*format: str = DEFAULT_FORMAT*) → list[Sequence[float]]

Returns all points as list of tuples, format specifies a user defined point format.

All points in *OCS* as (x, y) tuples (`dxf.elevation` is the z-axis value).

Parameters

format – format string, default is "xyseb", see *format codes*

set_points (*points: Iterable[Sequence[float]]*, *format: str = DEFAULT_FORMAT*) → None

Remove all points and append new *points*.

All coordinates in *OCS*.

Parameters

- **points** – iterable of point, point is (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, default is "xyseb", see *format codes*

points (*format: str = DEFAULT_FORMAT*) → Iterator[list[Sequence[float]]]

Context manager for polyline points. Returns a standard Python list of points, according to the format string.

All coordinates in *OCS*.

Parameters

format – format string, see *format codes*

transform (*m:* Matrix44) → LWPolyline

Transform the LWPOLYLINE entity by transformation matrix *m* inplace.

A non-uniform scaling is not supported if the entity contains circular arc segments (bulges).

Parameters

m – transformation `Matrix44`

Raises

NonUniformScalingError – for non-uniform scaling of entity containing circular arc segments (bulges)

virtual_entities () → Iterator[*Line* | *Arc*]

Yields the graphical representation of LWPOLYLINE as virtual DXF primitives (LINE or ARC).

These virtual entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode (*target_layout:* BaseLayout | *None = None*) → *EntityQuery*

Explode the LWPOLYLINE entity as DXF primitives (LINE or ARC) into the target layout, if the target layout is `None`, the target layout is the layout of the source entity. This method destroys the source entity.

Returns an `EntityQuery` container referencing all DXF primitives.

Parameters

target_layout – target layout for the DXF primitives, `None` for same layout as the source entity.

MLine

The MLINE entity (DXF Reference).

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'MLINE'
factory function	*add_mline()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**MLine**

dxf.**style_name**

MLineStyle name stored in Drawing.mline_styles dictionary, use *set_style()* to change the MLINESTYLE and update geometry accordingly.

dxf.**style_handle**

Handle of *MLineStyle*, use *set_style()* to change the MLINESTYLE and update geometry accordingly.

dxf.**scale_factor**

MLINE scaling factor, use method *set_scale_factor()* to change the scaling factor and update geometry accordingly.

dxf.**justification**

Justification defines the location of the MLINE in relation to the reference line, use method *set_justification()* to change the justification and update geometry accordingly.

Constants defined in *ezdxf.lldxf.const*:

dxf.justification	Value
MLINE_TOP	0
MLINE_ZERO	1
MLINE_BOTTOM	2
MLINE_RIGHT (alias)	0
MLINE_CENTER (alias)	1
MLINE_LEFT (alias)	2

dxf.**flags**

Use method *close()* and the properties *start_caps* and *end_caps* to change these flags.

Constants defined in *ezdxf.lldxf.const*:

dxf.flags	Value
MLINE_HAS_VERTEX	1
MLINE_CLOSED	2
MLINE_SUPPRESS_START_CAPS	4
MLINE_SUPPRESS_END_CAPS	8

dxf.**start_location**

Start location of the reference line. (read only)

dxf.**count**

Count of MLINE vertices. (read only)

dxf.**style_element_count**

Count of elements in *MLineStyle* definition. (read only)

dxf.**extrusion**

Normal vector of the entity plane, but MLINE is not an OCS entity, all vertices of the reference line are WCS! (read only)

vertices

MLINE vertices as *MLineVertex* objects, stored in a regular Python list.

property style: MLineStyle | None

Get associated MLINESTYLE.

set_style(*name: str*) → None

Set MLINESTYLE by name and update geometry accordingly. The MLINESTYLE definition must exist.

set_scale_factor(*value: float*) → None

Set the scale factor and update geometry accordingly.

set_justification(*value: int*) → None

Set MLINE justification and update geometry accordingly. See *dxf.justification* for valid settings.

property is_closed: bool

Returns `True` if MLINE is closed. Compatibility interface to *Polyline*

close(*state: bool = True*) → None

Get/set closed state of MLINE and update geometry accordingly. Compatibility interface to *Polyline*

property start_caps: bool

Get/Set start caps state. `True` to enable start caps and `False` tu suppress start caps.

property end_caps: bool

Get/Set end caps state. `True` to enable end caps and `False` tu suppress start caps.

__len__()

Count of MLINE vertices.

start_location() → *Vec3*

Returns the start location of the reference line. Callback function for *dxf.start_location*.

get_locations() → list[*ezdxf.math._vector.Vec3*]

Returns the vertices of the reference line.

extend(*vertices: Iterable[*UVec*]*) → None

Append multiple vertices to the reference line.

It is possible to work with 3D vertices, but all vertices have to be in the same plane and the normal vector of this plan is stored as extrusion vector in the MLINE entity.

clear() → None

Remove all MLINE vertices.

update_geometry() → None

Regenerate the MLINE geometry based on current settings.

generate_geometry(*vertices: list[*ezdxf.math._vector.Vec3*]*) → None

Regenerate the MLINE geometry for new reference line defined by *vertices*.

transform(*m:* Matrix44) → *DXFGraphic*

Transform MLINE entity by transformation matrix *m* inplace.

virtual_entities() → Iterator[DXFGraphic]

Yields virtual DXF primitives of the MLINE entity as LINE, ARC and HATCH entities.

These entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode(*target_layout:* BaseLayout | *None* = *None*) → *EntityQuery*

Explode the MLINE entity as LINE, ARC and HATCH entities into target layout, if target layout is `None`, the target layout is the layout of the MLINE. This method destroys the source entity.

Returns an *EntityQuery* container referencing all DXF primitives.

Parameters

target_layout – target layout for DXF primitives, `None` for same layout as source entity.

class ezdxf.entities.**MLineVertex**

location

Reference line vertex location.

line_direction

Reference line direction.

miter_direction

line_params

The line parameterization is a list of float values. The list may contain zero or more items.

The first value (miter-offset) is the distance from the vertex *location* along the *miter_direction* vector to the point where the line element's path intersects the miter vector.

The next value (line-start-offset) is the distance along the *line_direction* from the miter/line path intersection point to the actual start of the line element.

The next value (dash-length) is the distance from the start of the line element (dash) to the first break (gap) in the line element. The successive values continue to list the start and stop points of the line element in this segment of the mline.

fill_params

The fill parameterization is also a list of float values. Similar to the line parameterization, it describes the parameterization of the fill area for this mline segment. The values are interpreted identically to the line parameters and when taken as a whole for all line elements in the mline segment, they define the boundary of the fill area for the mline segment.

class ezdxf.entities.**MLineStyle**

The *MLineStyle* stores the style properties for the MLINE entity.

dxf.**name**

dxf.**description**

dxf.**flags**

dxf.**fill_color**

AutoCAD Color Index (ACI) value of the fill color

dxf.**start_angle**

dxf.**end_angle**

elements

MLineStyleElements object

update_all()

Update all MLINE entities using this MLINESTYLE.

The update is required if elements were added or removed or the offset of any element was changed.

class ezdxf.entities.mline.**MLineStyleElements**

elements

List of *MLineStyleElement* objects, one for each line element.

MLineStyleElements.**__len__**()

MLineStyleElements.**__getitem__**(*item*)

MLineStyleElements.**append**(*offset: float*, *color: int = 0*, *linetype: str = 'BYLAYER'*) → None

Append a new line element.

Parameters

- **offset** – normal offset from the reference line: if justification is MLINE_ZERO, positive values are above and negative values are below the reference line.
- **color** – *AutoCAD Color Index (ACI)* value
- **linetype** – linetype name

class ezdxf.entities.mline.**MLineStyleElement**

Named tuple to store properties of a line element.

offset

Normal offset from the reference line: if justification is MLINE_ZERO, positive values are above and negative values are below the reference line.

color

AutoCAD Color Index (ACI) value

linetype

Linetype name

Mesh

The MESH entity (DXF Reference) is a 3D surface in *WCS* build up from vertices and faces similar to the *Polyface* entity.

All vertices in *WCS* as (x, y, z) tuples

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'MESH'
Factory function	*ezdxf.layouts.BaseLayout.add_mesh()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

See also:

Tutorial for Mesh and helper classes: *MeshBuilder*, *MeshVertexMerger*

class ezdxf.entities.**Mesh**

dxf.**version**

dxf.**blend_crease**

0 = off, 1 = on

dxf.**subdivision_levels**

0 for no smoothing else integer greater than 0.

vertices

Vertices as list like *VertexArray*. (read/write)

edges

Edges as list like *TagArray*. (read/write)

faces

Faces as list like *TagList*. (read/write)

creases

Creases as array.array. (read/write)

edit_data() → Iterator[MeshData]

Context manager for various mesh data, returns a *MeshData* instance.

Despite that vertices, edge and faces are accessible as packed data types, the usage of *MeshData* by context manager *edit_data()* is still recommended.

transform(*m:* Matrix44) → Mesh

Transform the MESH entity by transformation matrix *m* inplace.

MeshData

class ezdxf.entities.**MeshData**

vertices

A standard Python list with (x, y, z) tuples (read/write)

faces

A standard Python list with (v1, v2, v3,…) tuples (read/write)

Each face consist of a list of vertex indices (= index in *vertices*).

edges

A Python list with (v1, v2) tuples (read/write). This list represents the edges to which the *edge_crease_values* values will be applied. Each edge consist of exact two vertex indices (= index in *vertices*).

edge_crease_values

A Python list of float values, one value for each edge. (read/write)

add_face (*vertices: Iterable[*UVec*]*) → Sequence[int]

Add a face by coordinates, vertices is a list of `(x, y, z)` tuples.

add_edge_crease (*v1: int*, *v2: int*, *crease: float*)

Add an edge crease value, the edge is defined by the vertex indices *v1* and *v2*. The crease value defines the amount of subdivision that will be applied to this edge. A crease value of the subdivision level prevents the edge from deformation and a value of 0.0 means no protection from subdividing.

optimize (*precision: int = 6*)

Try to reduce vertex count by merging near vertices. *precision* defines the decimal places for coordinate be equal to merge two vertices.

MPolygon

The MPOLYGON entity is not a core DXF entity and is not supported by all CAD applications and DXF libraries. The *MPolygon* class is very similar to the *Hatch* class with small differences in the supported features and DXF attributes.

The boundary paths of the MPOLYGON are visible and use the graphical DXF attributes of the main entity like `dxf.color`, `dxf.linetype` and so on. The solid filling is only visible if the attribute `dxf.solid_fill` is 1, the color of the solid fill is defined by `dxf.fill_color` as *AutoCAD Color Index (ACI)*. The MPOLYGON supports *ezdxf.entities.Gradient* settings like HATCH for DXF R2004 and newer. This feature is used by method *MPolygon.set_solid_fill()* to set a solid RGB fill color as linear gradient, this disables pattern fill automatically. The MPOLYGON does not support associated source path entities, because the MPOLYGON also represents the boundary paths as visible graphical objects. Hatch patterns are supported, but the hatch style tag is not supported, the default hatch style is `ezdxf.const.HATCH_STYLE_NESTED` and the style flags of the boundary paths are ignored. Background color for pattern fillings is supported, set background color by property *MPolygon.bgcolor* as RGB tuple.

Note: Background RGB fill color for solid fill and pattern fill is set differently!

Autodesk products do support polyline paths including bulges. An example for edge paths as boundary paths is not available or edge paths are not supported. *Ezdxf* does **not** export MPOLYGON entities including edge paths! The *BoundaryPaths.edge_to_polyline_paths()* method converts all edge paths to simple polyline paths with approximated curves, this conversion has to be done explicit.

See also:

For more information see the *ezdxf.entities.Hatch* documentation.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	`'MPOLYGON'`
Factory function	*ezdxf.layouts.BaseLayout.add_mpolygon()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

class ezdxf.entities.**MPolygon**

dxf.**pattern_name**

Pattern name as string

dxf.**solid_fill**

1	solid fill, better use: *MPolygon.set_solid_fill()*
0	pattern fill, better use: *MPolygon.set_pattern_fill()*

(search AutoCAD help for more information)

dxf.**pattern_type**

0	user
1	predefined
2	custom

dxf.**pattern_angle**

Actual pattern angle in degrees (float). Changing this value does not rotate the pattern, use *set_pattern_angle()* for this task.

dxf.**pattern_scale**

Actual pattern scaling factor (float). Changing this value does not scale the pattern use *set_pattern_scale()* for this task.

dxf.**pattern_double**

1 = double pattern size else 0. (int)

dxf.**elevation**

Z value represents the elevation height of the *OCS*. (float)

paths

BoundaryPaths object.

pattern

Pattern object.

gradient

Gradient object.

property has_solid_fill: bool

`True` if entity has a solid fill. (read only)

property has_pattern_fill: bool

`True` if entity has a pattern fill. (read only)

property has_gradient_data: bool

`True` if entity has a gradient fill. A hatch with gradient fill has also a solid fill. (read only)

property bgcolor: Tuple[int, int, int] | None

Set pattern fill background color as (r, g, b)-tuple, rgb values in the range [0, 255] (read/write/del)

usage:

```
r, g, b = entity.bgcolor  # get pattern fill background color
entity.bgcolor = (10, 20, 30)  # set pattern fill background color
del entity.bgcolor  # delete pattern fill background color
```

set_pattern_definition(*lines: Sequence*, *factor: float = 1*, *angle: float = 0*) → None

Setup pattern definition by a list of definition lines and the definition line is a 4-tuple (angle, base_point, offset, dash_length_items). The pattern definition should be designed for a pattern scale factor of 1 and a pattern rotation angle of 0.

- angle: line angle in degrees
- base-point: (x, y) tuple
- offset: (dx, dy) tuple
- dash_length_items: list of dash items (item > 0 is a line, item < 0 is a gap and item == 0.0 is a point)

Parameters

- **lines** – list of definition lines
- **factor** – pattern scale factor
- **angle** – rotation angle in degrees

set_pattern_scale(*scale: float*) → None

Sets the pattern scale factor and scales the pattern definition.

The method always starts from the original base scale, the `set_pattern_scale(1)` call resets the pattern scale to the original appearance as defined by the pattern designer, but only if the pattern attribute *dxf.pattern_scale* represents the actual scale, it cannot restore the original pattern scale from the pattern definition itself.

Parameters

scale – pattern scale factor

set_pattern_angle(*angle: float*) → None

Sets the pattern rotation angle and rotates the pattern definition.

The method always starts from the original base rotation of 0, the `set_pattern_angle(0)` call resets the pattern rotation angle to the original appearance as defined by the pattern designer, but only if the pattern attribute *dxf.pattern_angle* represents the actual pattern rotation, it cannot restore the original rotation angle from the pattern definition itself.

Parameters

angle – pattern rotation angle in degrees

set_solid_fill(*color: int = 7*, *style: int = 1*, *rgb: RGB | None = None*)

Set *MPolygon* to solid fill mode and removes all gradient and pattern fill related data.

Parameters

- **color** – *AutoCAD Color Index (ACI)*, (0 = BYBLOCK; 256 = BYLAYER)
- **style** – hatch style is not supported by MPOLYGON, just for symmetry to HATCH
- **rgb** – true color value as (r, g, b)-tuple - has higher priority than *color*. True color support requires DXF R2004+

set_pattern_fill(*name: str*, *color: int = 7*, *angle: float = 0.0*, *scale: float = 1.0*, *double: int = 0*, *style: int = 1*, *pattern_type: int = 1*, *definition=None*) → None

Sets the pattern fill mode and removes all gradient related data.

The pattern definition should be designed for a scale factor 1 and a rotation angle of 0 degrees. The predefined hatch pattern like "ANSI33" are scaled according to the HEADER variable $MEASUREMENT for ISO measurement (m, cm, …), or imperial units (in, ft, …), this replicates the behavior of BricsCAD.

Parameters

- **name** – pattern name as string
- **color** – pattern color as *AutoCAD Color Index (ACI)*
- **angle** – pattern rotation angle in degrees
- **scale** – pattern scale factor
- **double** – double size flag
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **pattern_type** – pattern type (0 = user-defined; 1 = predefined; 2 = custom)
- **definition** – list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], see `set_pattern_definition()`

set_gradient(*color1: Tuple[int, int, int] = (0, 0, 0)*, *color2: Tuple[int, int, int] = (255, 255, 255)*, *rotation: float = 0.0*, *centered: float = 0.0*, *one_color: int = 0*, *tint: float = 0.0*, *name: str = 'LINEAR'*) → None

Sets the gradient fill mode and removes all pattern fill related data, requires DXF R2004 or newer. A gradient filled hatch is also a solid filled hatch.

Valid gradient type names are:

- "LINEAR"
- "CYLINDER"
- "INVCYLINDER"
- "SPHERICAL"
- "INVSPHERICAL"
- "HEMISPHERICAL"
- "INVHEMISPHERICAL"
- "CURVED"
- "INVCURVED"

Parameters

- **color1** – (r, g, b)-tuple for first color, rgb values as int in the range [0, 255]
- **color2** – (r, g, b)-tuple for second color, rgb values as int in the range [0, 255]
- **rotation** – rotation angle in degrees
- **centered** – determines whether the gradient is centered or not
- **one_color** – 1 for gradient from *color1* to tinted *color1*
- **tint** – determines the tinted target *color1* for a one color gradient. (valid range 0.0 to 1.0)

- **name** – name of gradient type, default "LINEAR"

transform(*m:* Matrix44) → DXFPolygon

Transform entity by transformation matrix *m* inplace.

MText

The MTEXT entity (DXF Reference) fits a multiline text in a specified width but can extend vertically to an indefinite length. You can format individual words or characters within the *MText*.

See also:

Tutorial for MText and MTextEditor

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'MTEXT'
Factory function	*ezdxf.layouts.BaseLayout.add_mtext()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**MText**

dxf.**insert**

Insertion point (3D Point in *OCS*)

dxf.**char_height**

Initial text height (float); default=1.0

dxf.**width**

Reference text width (float), forces text wrapping at given width.

dxf.**attachment_point**

Constants defined in *ezdxf.lldxf.const*:

MText.dxf.attachment_point	Value
MTEXT_TOP_LEFT	1
MTEXT_TOP_CENTER	2
MTEXT_TOP_RIGHT	3
MTEXT_MIDDLE_LEFT	4
MTEXT_MIDDLE_CENTER	5
MTEXT_MIDDLE_RIGHT	6
MTEXT_BOTTOM_LEFT	7
MTEXT_BOTTOM_CENTER	8
MTEXT_BOTTOM_RIGHT	9

dxf.**flow_direction**

Constants defined in ezdxf.const:

MText.dxf.flow_direction	Value	Description
MTEXT_LEFT_TO_RIGHT	1	left to right
MTEXT_TOP_TO_BOTTOM	3	top to bottom
MTEXT_BY_STYLE	5	by style (the flow direction is inherited from the associated text style)

`dxf.`**`style`**

Text style (string); default is "STANDARD"

`dxf.`**`text_direction`**

X-axis direction vector in *WCS* (3D Point); default value is (1, 0, 0); if `dxf.rotation` and `dxf.text_direction` are present, `dxf.text_direction` wins.

`dxf.`**`rotation`**

Text rotation in degrees (float); default is 0

`dxf.`**`line_spacing_style`**

Line spacing style (int), see table below

`dxf.`**`line_spacing_factor`**

Percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00 (float).

Constants defined in *ezdxf.lldxf.const*:

MText.dxf.line_spacing_style	Value	Description
MTEXT_AT_LEAST	1	taller characters will override
MTEXT_EXACT	2	taller characters will not override

`dxf.`**`bg_fill`**

Defines the background fill type. (DXF R2007)

MText.dxf.bg_fill	Value	Description
MTEXT_BG_OFF	0	no background color
MTEXT_BG_COLOR	1	use specified color
MTEXT_BG_WINDOW_COLOR	2	use window color (?)
MTEXT_BG_CANVAS_COLOR	3	use canvas background color

`dxf.`**`box_fill_scale`**

Determines how much border there is around the text. (DXF R2007)

Requires that the attributes *bg_fill*, *bg_fill_color* are present otherwise AutoCAD complains.

It's recommended to use `set_bg_color()`

`dxf.`**`bg_fill_color`**

Background fill color as *AutoCAD Color Index (ACI)* (DXF R2007)

It's recommended to use `set_bg_color()`

`dxf.`**`bg_fill_true_color`**

Background fill color as true color value (DXF R2007), also the `dxf.bg_fill_color` attribute must be present otherwise AutoCAD complains.

It's recommended to use `set_bg_color()`

`dxf.`**`bg_fill_color_name`**

Background fill color as name string (?) (DXF R2007), also the `dxf.bg_fill_color` attribute must be present otherwise AutoCAD complains.

It's recommended to use `set_bg_color()`

`dxf.`**`transparency`**

Transparency of background fill color (DXF R2007), not supported by AutoCAD nor BricsCAD.

`text`

MTEXT content as string (read/write).

The line ending character `\n` will be replaced by the MTEXT line ending `\P` at DXF export, but **not** vice versa the `\P` character by `\n` at DXF file loading, therefore loaded MTEXT entities always use the `\P` character for line endings.

`set_location`(*insert:* UVec, *rotation: float | None = None*, *attachment_point: int | None = None*) → MText

Sets the attributes *`dxf.insert`*, *`dxf.rotation`* and *`dxf.attachment_point`*, `None` for *`dxf.rotation`* or *`dxf.attachment_point`* preserves the existing value.

`get_rotation`() → float

Returns the text rotation in degrees.

`set_rotation`(*angle: float*) → MText

Sets the attribute `rotation` to *angle* (in degrees) and deletes *`dxf.text_direction`* if present.

`get_text_direction`() → *Vec3*

Returns the horizontal text direction as *`Vec3`* object, even if only the text rotation is defined.

`set_bg_color`(*color: int | str | Tuple[int, int, int] | None*, *scale: float = 1.5*, *text_frame=False*)

Sets the background color as *AutoCAD Color Index (ACI)* value, as name string or as (r, g, b) tuple.

Use the special color name `canvas`, to set the background color to the canvas background color. Remove the background filling by setting argument *color* to `None`.

Parameters

- **`color`** – color as *AutoCAD Color Index (ACI)*, string, (r, g, b) tuple or `None`
- **`scale`** – determines how much border there is around the text, the value is based on the text height, and should be in the range of [1, 5], where 1 fits exact the MText entity.
- **`text_frame`** – draw a text frame in text color if `True`

`__iadd__`(*text: str*) → MText

Append *text* to existing content (*`text`* attribute).

`append`(*text: str*) → MText

Append *text* to existing content (*`text`* attribute).

`plain_text`(*split=False*, *fast=True*) → list[str] | str

Returns the text content without inline formatting codes.

The "fast" mode is accurate if the DXF content was created by reliable (and newer) CAD applications like AutoCAD or BricsCAD. The "accurate" mode is for some rare cases where the content was created by older CAD applications or unreliable DXF libraries and CAD applications.

Parameters

- **split** – split content text at line breaks if `True` and returns a list of strings without line endings
- **fast** – uses the "fast" mode to extract the plain MTEXT content if `True` or the "accurate" mode if set to `False`

all_columns_plain_text(*split=False*) → list[str] | str

Returns the text content of all columns without inline formatting codes.

Parameters

split – split content text at line breaks if `True` and returns a list of strings without line endings

all_columns_raw_content() → str

Returns the text content of all columns as a single string including the inline formatting codes.

transform(*m:* Matrix44) → MText

Transform the MTEXT entity by transformation matrix *m* inplace.

ucs() → *UCS*

Returns the *UCS* of the *MText* entity, defined by the insert location (origin), the text direction or rotation (x-axis) and the extrusion vector (z-axis).

MText Inline Codes

Code	Description
\L	Start underline
\l	Stop underline
\O	Start overline
\o	Stop overline
\K	Start strike-through
\k	Stop strike-through
\P	New paragraph (new line)
\p	Paragraphs properties: indentation, alignment, tabulator stops
\X	Paragraph wrap on the dimension line (only in dimensions)
\Q	Slanting (oblique) text by angle - e.g. \Q30;
\H	Text height - e.g. relative \H3x; absolut \H3;
\W	Text width - e.g. relative \W0.8x; absolut \W0.8;
\T	Tracking, character spacing - e.g. relative \T0.5x; absolut \T2;
\F	Font selection e.g. \Fgdt;o - GDT-tolerance
\S	Stacking, fractions e.g. \SA^ B; space after "^" is required to avoid caret decoding, \SX/Y; \S1#4;
\A	Alignment • \A0; = bottom • \A1; = center • \A2; = top
\C	Color change • \C1; = red • \C2; = yellow • \C3; = green • \C4; = cyan • \C5; = blue • \C6; = magenta • \C7; = white
\~	Non breaking space
{ }	Braces - define the text area influenced by the code, codes and braces can be nested up to 8 levels deep
\	Escape character - e.g. \{ = "{"

Convenient constants defined in MTextEditor:

Constant	Description
UNDERLINE_START	start underline text
UNDERLINE_STOP	stop underline text
OVERSTRIKE_START	start overline
OVERSTRIKE_STOP	stop overline
STRIKE_START	start strike through
STRIKE_STOP	stop strike through
GROUP_START	start of group
GROUP_END	end of group
NEW_LINE	start in new line
NBSP	none breaking space

MultiLeader

The MULTILEADER entity (DXF Reference) represents one or more leaders, made up of one or more vertices (or spline fit points) and an arrowhead. In contrast to the `Leader` entity the text- or block content is part of the MULTILEADER entity.

AutoCAD, BricsCAD and maybe other CAD applications do accept "MLEADER" as type string but they always create entities with "MULTILEADER" as type string.

Because of the complexity of the MULTILEADER entity, the usage of factory methods to create new entities by special builder classes is recommended:

- `add_multileader_mtext()` returns a new `MultiLeaderMTextBuilder` instance
- `add_multileader_block()` returns a new `MultiLeaderBlockBuilder` instance

The visual design is based on an associated `MLeaderStyle`, but almost all attributes are also stored in the MULTILEADER entity itself.

The attribute `MultiLeader.dxf.property_override_flags` should indicate which MLEADERSTYLE attributes are overridden by MULTILEADER attributes, but these flags do not always reflect the state of overridden attributes. The *ezdxf* MULTILEADER renderer uses always the attributes from the MULTILEADER entity and ignores the override flags.

All vertices are WCS coordinates, even those for BLOCK entities which are OCS coordinates for regular usage.

See also:

- `ezdxf.entities.MLeaderStyle`
- `ezdxf.render.MultiLeaderBuilder`
- *Tutorial for MultiLeader*
- *MULTILEADER Internals*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'MULTILEADER'
Factory functions	• *ezdxf.layouts.BaseLayout.add_multileader_mtext()* • *ezdxf.layouts.BaseLayout.add_multileader_block()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**MultiLeader**

dxf.**arrow_head_handle**
handle of the arrow head, see also *ezdxf.render.arrows* module, "closed filled" arrow if not set

dxf.**arrow_head_size**
arrow head size in drawing units

dxf.**block_color**
block color as *raw-color* value, default is BY_BLOCK_RAW_VALUE

dxf.**block_connection_type**

0	center extents
1	insertion point

dxf.**block_record_handle**
handle to block record of the BLOCK content

dxf.**block_rotation**
BLOCK rotation in radians

dxf.**block_scale_vector**
Vec3 object which stores the scaling factors for the x-, y- and z-axis

dxf.**content_type**

0	none
1	BLOCK
2	MTEXT
3	TOLERANCE

dxf.**dogleg_length**
dogleg length in drawing units

dxf.**has_dogleg**

dxf.**has_landing**

dxf.**has_text_frame**

dxf.**is_annotative**

dxf.**is_text_direction_negative**

dxf.**leader_extend_to_text**

dxf.**leader_line_color**

leader line color as *raw-color* value

dxf.**leader_linetype_handle**

handle of the leader linetype, "CONTINUOUS" if not set

dxf.**leader_lineweight**

dxf.**leader_type**

0	invisible
1	straight line leader
2	spline leader

dxf.**property_override_flags**

Each bit shows if the MLEADERSTYLE is overridden by the value in the MULTILEADER entity, but this is not always the case for all values, it seems to be save to always use the value from the MULTILEADER entity.

dxf.**scale**

overall scaling factor

dxf.**style_handle**

handle to the associated MLEADERSTYLE object

dxf.**text_IPE_align**

unknown meaning

dxf.**text_alignment_type**

unknown meaning - its not the MTEXT attachment point!

dxf.**text_angle_type**

0	text angle is equal to last leader line segment angle
1	text is horizontal
2	text angle is equal to last leader line segment angle, but potentially rotated by 180 degrees so the right side is up for readability.

dxf.**text_attachment_direction**

defines whether the leaders attach to the left & right of the content BLOCK/MTEXT or attach to the top & bottom:

0	horizontal - left & right of content
1	vertical - top & bottom of content

dxf.**text_attachment_point**

MTEXT attachment point

1	top left
2	top center
3	top right

dxf.**text_bottom_attachment_type**

9	center
10	overline and center

dxf.**text_color**

MTEXT color as *raw-color* value

dxf.**text_left_attachment_type**

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

dxf.**text_right_attachment_type**

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

dxf.**text_style_handle**

handle of the MTEXT text style, “Standard” if not set

dxf.**text_top_attachment_type**

9	center
10	overline and center

`dxf.`**version**

always 2?

context

MLeaderContext instance

arrow_heads

list of *ArrowHeadData*

block_attribs

list of *AttribData*

property has_mtext_content: bool

`True` if MULTILEADER has MTEXT content.

get_mtext_content() → str

Get MTEXT content as string, return "" if MULTILEADER has BLOCK content.

set_mtext_content(*text: str*)

Set MTEXT content as string, does nothing if MULTILEADER has BLOCK content.

property has_block_content: bool

`True` if MULTILEADER has BLOCK content.

get_block_content() → dict[str, str]

Get BLOCK attributes as dictionary of (tag, value) pairs. Returns an empty dictionary if MULTILEADER has MTEXT content.

set_block_content(*content: dict[str, str]*)

Set BLOCK attributes by a dictionary of (tag, value) pairs. Does nothing if MULTILEADER has MTEXT content.

virtual_entities() → Iterator[DXFGraphic]

Yields the graphical representation of MULTILEADER as virtual DXF primitives.

These entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode(*target_layout:* BaseLayout | *None* = *None*) → *EntityQuery*

Explode MULTILEADER as DXF primitives into target layout, if target layout is `None`, the target layout is the layout of the source entity.

Returns an *EntityQuery* container with all DXF primitives.

Parameters

target_layout – target layout for the DXF primitives, `None` for same layout as the source entity.

transform(*m:* Matrix44) → MultiLeader

Transform the MULTILEADER entity by transformation matrix *m* inplace.

Non-uniform scaling is not supported.

Parameters

m – transformation *Matrix44*

Raises

NonUniformScalingError – for non-uniform scaling

class ezdxf.entities.**MLeaderContext**

leaders
list of *LeaderData* objects

scale
redundant data: *MultiLeader.dxf.scale*

base_point
insert location as `Vec3` of the MTEXT or the BLOCK entity?

char_height
MTEXT char height, already scaled

arrow_head_size
redundant data: *MultiLeader.dxf.arrow_head_size*

landing_gap_size

left_attachment
redundant data: *MultiLeader.dxf.text_left_attachment_type*

right_attachment
redundant data: *MultiLeader.dxf.text_right_attachment_type*

text_align_type
redundant data: *MultiLeader.dxf.text_attachment_point*

attachment_type
BLOCK alignment?

0	content extents
1	insertion point

mtext
instance of *MTextData* if content is MTEXT otherwise `None`

block
instance of *BlockData* if content is BLOCK otherwise `None`

plane_origin
`Vec3`

plane_x_axis
`Vec3`

plane_y_axis
`Vec3`

plane_normal_reversed
the plan normal is x-axis "cross" y-axis (right-hand-rule), this flag indicates to invert this plan normal

top_attachment
redundant data: *MultiLeader.dxf.text_top_attachment_type*

bottom_attachment
redundant data: *MultiLeader.dxf.text_bottom_attachment_type*

class ezdxf.entities.**LeaderData**

lines
list of *LeaderLine*

has_last_leader_line
unknown meaning

has_dogleg_vector

last_leader_point
WCS point as `Vec3`

dogleg_vector
WCS direction as `Vec3`

dogleg_length
redundant data: *MultiLeader.dxf.dogleg_length*

index
leader index?

attachment_direction
redundant data: *MultiLeader.dxf.text_attachment_direction*

breaks
list of break vertices as `Vec3` objects

class ezdxf.entities.**LeaderLine**

vertices
list of WCS coordinates as `Vec3`

breaks
mixed list of mixed integer indices and break coordinates or `None` leader lines without breaks in it

index
leader line index?

color
leader line color override, ignore override value if BY_BLOCK_RAW_VALUE

class ezdxf.entities.**ArrowHeadData**

index
arrow head index?

handle
handle to arrow head block

class ezdxf.entities.**AttribData**

handle
handle to `Attdef` entity in the BLOCK definition

index
unknown meaning

width
text width factor?

text
Attrib content

class ezdxf.entities.**MTextData**
stores the content and attributes of the MTEXT entity

default_content
content as string

extrusion
extrusion vector of the MTEXT entity but MTEXT is not an OCS entity!

style_handle
redundant data: *MultiLeader.dxf.text_style_handle*

insert
insert location in WCS coordinates, same as *MLeaderContext.base_point*?

text_direction
"horizontal" text direction vector in WCS

rotation
rotation angle in radians (!) around the extrusion vector, calculated as it were an OCS entity

width
unscaled column width

defined_height
unscaled defined column height

line_spacing_factor
see *MText.dxf.line_spacing_factor*

line_spacing_style
see *MText.dxf.line_spacing_style*

color
redundant data: *MultiLeader.dxf.text_color*

alignment
redundant data: *MultiLeader.dxf.text_attachment_point*

flow_direction

1	horizontal
3	vertical
6	by text style

bg_color
background color as *raw-color* value

bg_scale_factor
see *MText.dxf.box_fill_scale*

bg_transparency
background transparency value

use_window_bg_color

has_bg_fill

column_type
unknown meaning - most likely:

0	none
1	static
2	dynamic

use_auto_height

column_width
unscaled column width, redundant data *width*

column_gutter_width
unscaled column gutter width

column_flow_reversed

column_sizes
list of unscaled columns heights for dynamic column with manual heights

use_word_break

class ezdxf.entities.**BlockData**
stores the attributes for the *Insert* entity

block_record_handle
redundant data: *MultiLeader.dxf.block_record_handle*

extrusion
extrusion vector in WCS

insert
insertion location in WCS as Vec3, same as *MLeaderContext.base_point*?

scale
redundant data: *MultiLeader.dxf.block_scale_vector*

rotation
redundant data: *MultiLeader.dxf.block_rotation*

color
redundant data: *MultiLeader.dxf.block_color*

Point

The POINT entity (DXF Reference) represents a dimensionless point in *WCS*.

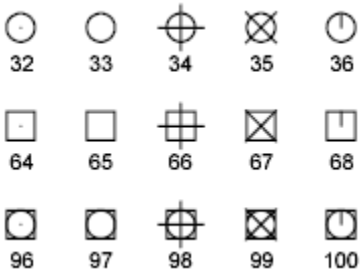
The POINT styling is a global setting, stored as header variable $PDMODE, this also means **all** POINT entities in a DXF document have the same styling:

0	center dot (.)
1	none ( )
2	cross (+)
3	x-cross (x)
4	tick (‘)

Combined with these bit values

32	circle
64	Square

e.g. circle + square + center dot = 32 + 64 + 0 = 96



The size of the points is defined by the header variable $PDSIZE:

0	5% of draw area height
<0	Specifies a percentage of the viewport size
>0	Specifies an absolute size

See also:

- *Tutorial for Simple DXF Entities*, section *Point*

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	`'POINT'`
Factory function	*ezdxf.layouts.BaseLayout.add_point()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Point**

`dxf.location`

Location of the point (2D/3D Point in *WCS*)

`dxf.angle`

Angle in degrees of the x-axis for the UCS in effect when POINT was drawn (float); used when PDMODE is nonzero.

transform (*m:* Matrix44) → Point

Transform the POINT entity by transformation matrix *m* inplace.

translate (*dx: float*, *dy: float*, *dz: float*) → Point

Optimized POINT translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

virtual_entities (*pdsize: float = 1*, *pdmode: int = 0*) → Iterator[DXFGraphic]

Yields the graphical representation of POINT as virtual DXF primitives (LINE and CIRCLE). The dimensionless point is rendered as zero-length line!

Check for this condition:

```
e.dxftype() == 'LINE' and e.dxf.start.isclose(e.dxf.end)
```

if the rendering engine can't handle zero-length lines.

Parameters

- **pdsize** – point size in drawing units
- **pdmode** – point styling mode

Polyline

The POLYLINE entity (POLYLINE DXF Reference) is very complex, it's used to build 2D/3D polylines, 3D meshes and 3D polyfaces. For every type exists a different wrapper class but they all have the same DXF type "POLYLINE". Detect the actual POLYLINE type by the method *Polyline.get_mode()*.

POLYLINE types returned by *Polyline.get_mode()*:

- `'AcDb2dPolyline'` for 2D *Polyline*
- `'AcDb3dPolyline'` for 3D *Polyline*
- `'AcDbPolygonMesh'` for *Polymesh*
- `'AcDbPolyFaceMesh'` for *Polyface*

For 2D entities all vertices in *OCS*.

For 3D entities all vertices in *WCS*.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	`'POLYLINE'`
2D factory function	*ezdxf.layouts.BaseLayout.add_polyline2d()*
3D factory function	*ezdxf.layouts.BaseLayout.add_polyline3d()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Polyline**

The *Vertex* entities are stored in the Python list *Polyline.vertices*. The VERTEX entities can be retrieved and deleted by direct access to the *Polyline.vertices* attribute:

```
# delete first and second vertex
del polyline.vertices[:2]
```

dxf.**elevation**

Elevation point, the X and Y values are always 0, and the Z value is the polyline elevation (3D Point).

dxf.**flags**

Constants defined in *ezdxf.lldxf.const*:

Polyline.dxf.flags	Value	Description
POLYLINE_CLOSED	1	This is a closed Polyline (or a polygon mesh closed in the M direction)
POLY-LINE_MESH_CLOSED_M_DIRECT	1	equals POLYLINE_CLOSED
POLY-LINE_CURVE_FIT_VERTICES_AD	2	Curve-fit vertices have been added
POLY-LINE_SPLINE_FIT_VERTICES_AD	4	Spline-fit vertices have been added
POLYLINE_3D_POLYLINE	8	This is a 3D Polyline
POLYLINE_3D_POLYMESH	16	This is a 3D polygon mesh
POLY-LINE_MESH_CLOSED_N_DIRECT	32	The polygon mesh is closed in the N direction
POLYLINE_POLYFACE_MESH	64	This Polyline is a polyface mesh
POLY-LINE_GENERATE_LINETYPE_PA	128	The linetype pattern is generated continuously around the vertices of this Polyline

dxf.**default_start_width**

Default line start width (float); default is 0

dxf.**default_end_width**

Default line end width (float); default is 0

dxf.**m_count**

Polymesh M vertex count (int); default is 1

dxf.**n_count**

Polymesh N vertex count (int); default is 1

dxf.**m_smooth_density**

Smooth surface M density (int); default is 0

dxf.**n_smooth_density**

Smooth surface N density (int); default is 0

dxf.**smooth_type**

Curves and smooth surface type (int); default is 0, see table below

Constants for smooth_type defined in *ezdxf.lldxf.const*:

Polyline.dxf.smooth_type	Value	Description
POLYMESH_NO_SMOOTH	0	no smooth surface fitted
POLYMESH_QUADRATIC_BSPLINE	5	quadratic B-spline surface
POLYMESH_CUBIC_BSPLINE	6	cubic B-spline surface
POLYMESH_BEZIER_SURFACE	8	Bezier surface

vertices

List of *Vertex* entities.

is_2d_polyline

`True` if POLYLINE is a 2D polyline.

is_3d_polyline

`True` if POLYLINE is a 3D polyline.

is_polygon_mesh

`True` if POLYLINE is a polygon mesh, see *Polymesh*

is_poly_face_mesh

`True` if POLYLINE is a poly face mesh, see *Polyface*

is_closed

`True` if POLYLINE is closed.

is_m_closed

`True` if POLYLINE (as *Polymesh*) is closed in m direction.

is_n_closed

`True` if POLYLINE (as *Polymesh*) is closed in n direction.

has_arc

Returns `True` if 2D POLYLINE has an arc segment.

has_width

Returns `True` if 2D POLYLINE has default width values or any segment with width attributes.

get_mode() → str

Returns POLYLINE type as string:

- "AcDb2dPolyline"
- "AcDb3dPolyline"
- "AcDbPolygonMesh"
- "AcDbPolyFaceMesh"

m_close(*status=True*) → None

Close POLYMESH in m direction if *status* is `True` (also closes POLYLINE), clears closed state if *status* is `False`.

n_close(*status=True*) → None

Close POLYMESH in n direction if *status* is `True`, clears closed state if *status* is `False`.

close(*m_close=True*, *n_close=False*) → None

Set closed state of POLYMESH and POLYLINE in m direction and n direction. `True` set closed flag, `False` clears closed flag.

__len__() → int

Returns count of *Vertex* entities.

__getitem__(*pos*) → DXFVertex

Get *Vertex* entity at position *pos*, supports list-like slicing.

points() → Iterator[*Vec3*]

Returns iterable of all polyline vertices as (x, y, z) tuples, not as *Vertex* objects.

append_vertex(*point:* UVec, *dxfattribs=None*) → None

Append a single *Vertex* entity at location *point*.

Parameters

- **point** – as (x, y[, z]) tuple
- **dxfattribs** – dict of DXF attributes for *Vertex* class

append_vertices(*points: Iterable[*UVec*]*, *dxfattribs=None*) → None

Append multiple *Vertex* entities at location *points*.

Parameters

- **points** – iterable of (x, y[, z]) tuples
- **dxfattribs** – dict of DXF attributes for the VERTEX objects

append_formatted_vertices(*points: Iterable[*UVec*]*, *format: str = 'xy'*, *dxfattribs=None*) → None

Append multiple *Vertex* entities at location *points*.

Parameters

- **points** – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, default is "xy", see: *User Defined Point Format Codes*
- **dxfattribs** – dict of DXF attributes for the VERTEX objects

insert_vertices(*pos: int*, *points: Iterable[*UVec*]*, *dxfattribs=None*) → None

Insert vertices *points* into *Polyline.vertices* list at insertion location *pos* .

Parameters

- **pos** – insertion position of list *Polyline.vertices*
- **points** – list of (x, y[, z]) tuples
- **dxfattribs** – dict of DXF attributes for *Vertex* class

transform(*m:* Matrix44) → Polyline

Transform the POLYLINE entity by transformation matrix *m* inplace.

A non-uniform scaling is not supported if a 2D POLYLINE contains circular arc segments (bulges).

Parameters

m – transformation *Matrix44*

Raises

NonUniformScalingError – for non-uniform scaling of 2D POLYLINE containing circular arc segments (bulges)

virtual_entities() → Iterator[*Line* | *Arc* | *Face3d*]

Yields the graphical representation of POLYLINE as virtual DXF primitives (LINE, ARC or 3DFACE).

These virtual entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode(*target_layout:* BaseLayout | *None* = *None*) → *EntityQuery*

Explode the POLYLINE entity as DXF primitives (LINE, ARC or 3DFACE) into the target layout, if the target layout is `None`, the target layout is the layout of the POLYLINE entity.

Returns an *EntityQuery* container referencing all DXF primitives.

Parameters

target_layout – target layout for DXF primitives, `None` for same layout as source entity.

Vertex

A VERTEX (VERTEX DXF Reference) represents a polyline/mesh vertex.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	`'VERTEX'`
Factory function	*Polyline.append_vertex()*
Factory function	`Polyline.extend()`
Factory function	*Polyline.insert_vertices()*
Inherited DXF Attributes	*Common graphical DXF attributes*

class ezdxf.entities.**Vertex**

dxf.**location**

Vertex location (2D/3D Point *OCS* when 2D, *WCS* when 3D)

dxf.**start_width**

Line segment start width (float); default is 0

dxf.**end_width**

Line segment end width (float); default is 0

dxf.**bulge**

Bulge value (float); default is 0.

The bulge value is used to create arc shaped line segments.

dxf.**flags**

Constants defined in *ezdxf.lldxf.const*:

Vertex.dxf.flags	Valu	Description
VTX_EXTRA_VERTE	1	Extra vertex created by curve-fitting
VTX_CURVE_FIT_TA	2	curve-fit tangent defined for this vertex. A curve-fit tangent direction of 0 may be omitted from the DXF output, but is significant if this bit is set.
VTX_SPLINE_VERTE	8	spline vertex created by spline-fitting
VTX_SPLINE_FRAME	16	spline frame control point
VTX_3D_POLYLINE_	32	3D polyline vertex
VTX_3D_POLYGON_M	64	3D polygon mesh
VTX_3D_POLYFACE_	128	polyface mesh vertex

dxf.**tangent**
Curve fit tangent direction (float), used for 2D spline in DXF R12.

dxf.**vtx1**
Index of 1st vertex, if used as face (feature for experts)

dxf.**vtx2**
Index of 2nd vertex, if used as face (feature for experts)

dxf.**vtx3**
Index of 3rd vertex, if used as face (feature for experts)

dxf.**vtx4**
Index of 4th vertex, if used as face (feature for experts)

is_2d_polyline_vertex

is_3d_polyline_vertex

is_polygon_mesh_vertex

is_poly_face_mesh_vertex

is_face_record

format(*format='xyz'*) → Sequence
Return formatted vertex components as tuple.

Format codes:

- "x" = x-coordinate
- "y" = y-coordinate
- "z" = z-coordinate
- "s" = start width
- "e" = end width
- "b" = bulge value
- "v" = (x, y, z) as tuple

Args:
format: format string, default is "xyz"

Polymesh

Subclass of	*ezdxf.entities.Polyline*
DXF type	'POLYLINE'
Factory function	*ezdxf.layouts.BaseLayout.add_polymesh()*
Inherited DXF Attributes	*Common graphical DXF attributes*

class ezdxf.entities.**Polymesh**
A polymesh is a grid of m_count by n_count vertices, every vertex has its own (x, y, z) location. The *Polymesh* is a subclass of *Polyline*, the DXF type is also "POLYLINE", the method get_mode() returns "AcDbPolygonMesh".

get_mesh_vertex(*pos: tuple[int, int]*) → DXFVertex

Get location of a single mesh vertex.

Parameters

pos – 0-based (row, col) tuple, position of mesh vertex

set_mesh_vertex(*pos: tuple[int, int]*, *point:* UVec, *dxfattribs=None*)

Set location and DXF attributes of a single mesh vertex.

Parameters

- **pos** – 0-based (row, col) tuple, position of mesh vertex
- **point** – (x, y, z) tuple, new 3D coordinates of the mesh vertex
- **dxfattribs** – dict of DXF attributes

get_mesh_vertex_cache() → MeshVertexCache

Get a *MeshVertexCache* object for this POLYMESH. The caching object provides fast access to the `location` attribute of mesh vertices.

MeshVertexCache

class ezdxf.entities.**MeshVertexCache**

Cache mesh vertices in a dict, keys are 0-based (row, col) tuples.

Set vertex location: `cache[row, col] = (x, y, z)`

Get vertex location: `x, y, z = cache[row, col]`

vertices

Dict of mesh vertices, keys are 0-based (row, col) tuples.

__getitem__(*pos: tuple[int, int]*) → *UVec*

Get mesh vertex location as (x, y, z)-tuple.

Parameters

pos – 0-based (row, col)-tuple.

__setitem__(*pos: tuple[int, int]*, *location:* UVec) → None

Get mesh vertex location as (x, y, z)-tuple.

Parameters

- **pos** – 0-based (row, col)-tuple.
- **location** – (x, y, z)-tuple

Polyface

Subclass of	*ezdxf.entities.Polyline*
DXF type	'POLYLINE'
Factory function	*ezdxf.layouts.BaseLayout.add_polyface()*
Inherited DXF Attributes	*Common graphical DXF attributes*

See also:

Tutorial for Polyface

class ezdxf.entities.**Polyface**

A polyface consist of multiple 3D areas called faces, only faces with 3 or 4 vertices are supported. The *Polyface* is a subclass of *Polyline*, the DXF type is also "POLYLINE", the *get_mode()* returns "AcDbPolyFaceMesh".

append_face(*face: FaceType*, *dxfattribs=None*) → None

Append a single face. A *face* is a sequence of (x, y, z) tuples.

Parameters

- **face** – sequence of (x, y, z) tuples
- **dxfattribs** – dict of DXF attributes for the VERTEX objects

append_faces(*faces: Iterable[FaceType]*, *dxfattribs=None*) → None

Append multiple *faces*. *faces* is a list of single faces and a single face is a sequence of (x, y, z) tuples.

Parameters

- **faces** – iterable of sequences of (x, y, z) tuples
- **dxfattribs** – dict of DXF attributes for the VERTEX entity

faces() → Iterator[list[ezdxf.entities.polyline.DXFVertex]]

Iterable of all faces, a face is a tuple of vertices.

Returns
list of [vertex, vertex, vertex, [vertex,] face_record]

optimize(*precision: int = 6*) → None

Rebuilds the *Polyface* by merging vertices with nearly same vertex locations.

Parameters
precision – floating point precision for determining identical vertex locations

Ray

The RAY entity (DXF Reference) starts at `Ray.dxf.point` and continues to infinity (construction line).

Subclass of	*ezdxf.entities.XLine*
DXF type	`'RAY'`
Factory function	*ezdxf.layouts.BaseLayout.add_ray()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

class ezdxf.entities.**Ray**

dxf.**start**

Start point as (3D Point in *WCS*)

dxf.**unit_vector**

Unit direction vector as (3D Point in *WCS*)

transform(*m:* Matrix44) → XLine

Transform the XLINE/RAY entity by transformation matrix *m* inplace.

translate(*dx: float*, *dy: float*, *dz: float*) → XLine

Optimized XLINE/RAY translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

Region

REGION entity (DXF Reference) created by an ACIS geometry kernel provided by the Spatial Corp.

See also:

Ezdxf has only very limited support for ACIS based entities, for more information see the FAQ: *How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?*

Subclass of	*ezdxf.entities.Body*
DXF type	'REGION'
Factory function	*ezdxf.layouts.BaseLayout.add_region()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Region**

Same attributes and methods as parent class *Body*.

Shape

The SHAPE entity (DXF Reference) is used like a block references, each SHAPE reference can be scaled and rotated individually. The SHAPE definitions are stored in external shape files (*.SHX), and *ezdxf* can not load or create these shape files.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'SHAPE'
Factory function	*ezdxf.layouts.BaseLayout.add_shape()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Shape**

dxf.**insert**

Insertion location as (2D/3D Point in *WCS*)

dxf.**name**

Shape name (str)

dxf.**size**

Shape size (float)

`dxf.`**`rotation`**

Rotation angle in degrees; default value is 0

`dxf.`**`xscale`**

Relative X scale factor (float); default value is 1

`dxf.`**`oblique`**

Oblique angle in degrees (float); default value is 0

`transform`(*m:* Matrix44) → *Shape*

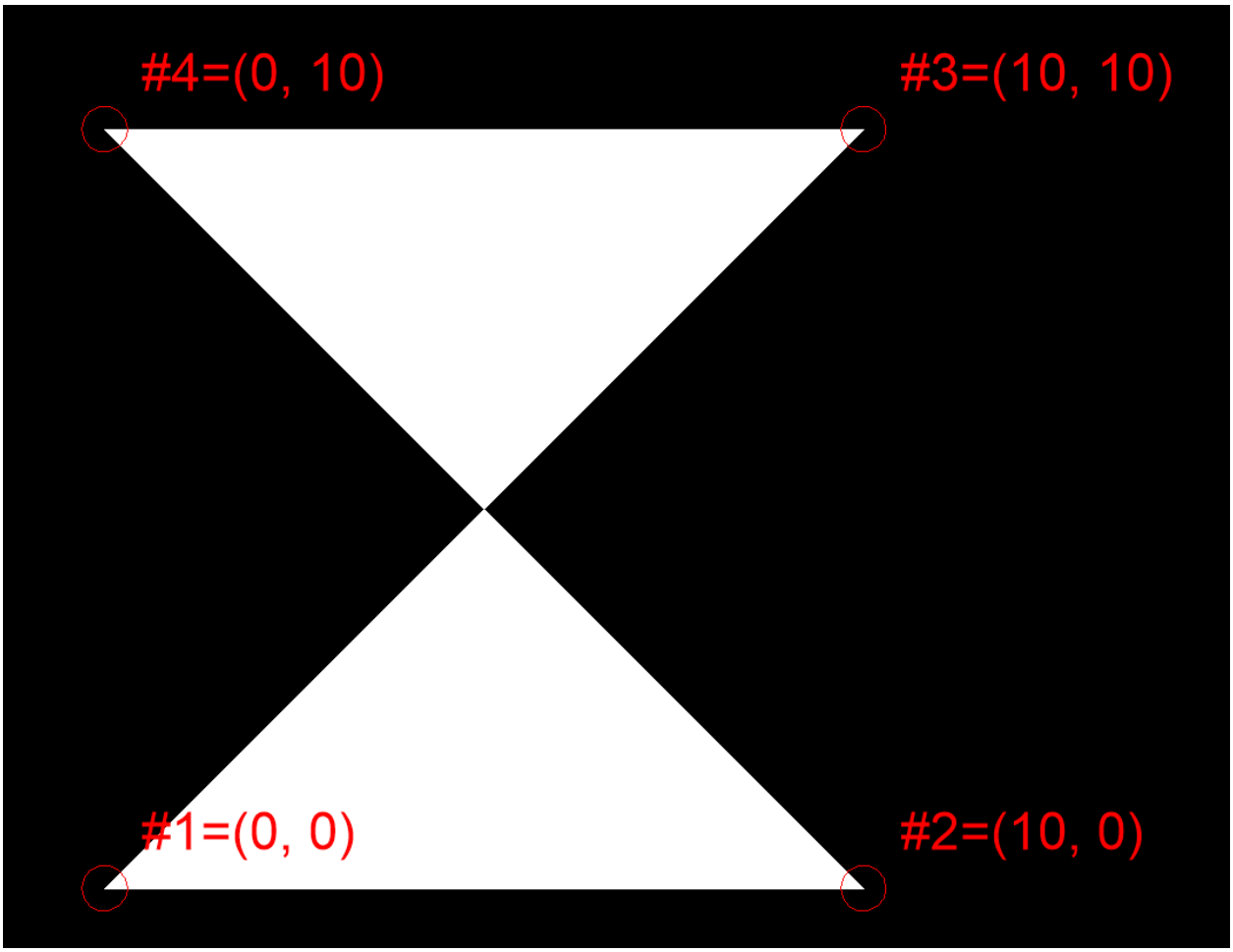
Transform the SHAPE entity by transformation matrix *m* inplace.

Solid

The SOLID entity (DXF Reference) is a filled triangle or quadrilateral. Access vertices by name (`entity.dxf.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`). If only 3 vertices are provided the last (3rd) vertex will be repeated in the DXF file.

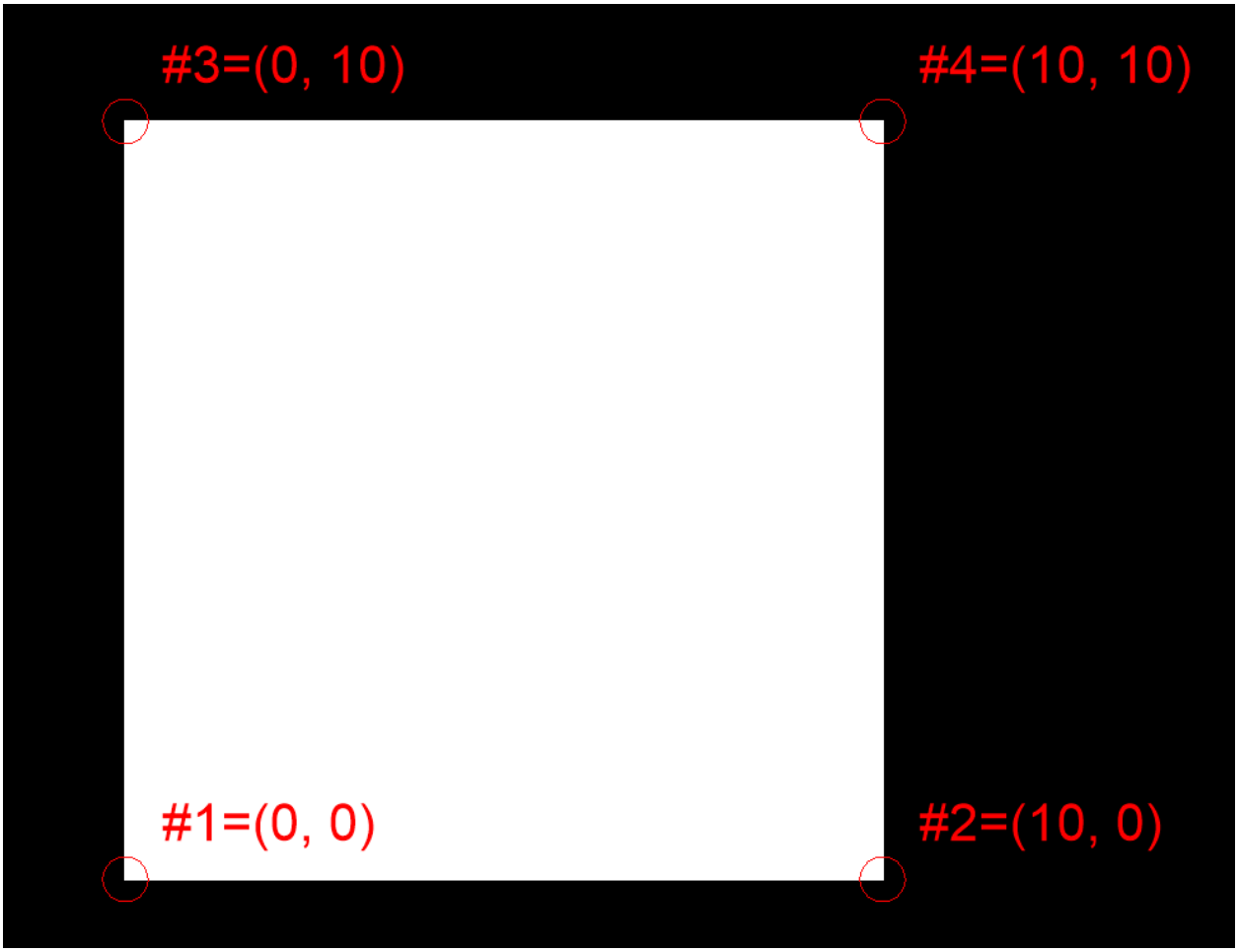
The SOLID entity stores the vertices in an unusual way, the last two vertices are reversed:

```
msp.add_solid([(0, 0), (10, 0), (10, 10), (0, 10)])
```



Reverse the last two vertices to get the *expected* square:

```
msp.add_solid([(0, 0), (10, 0), (0, 10), (10, 10)])
```



Note: The quirky vertex order is preserved at the lowest access level because *ezdxf* is intended as a DXF file format interface and presents the content of the DXF document to the package user as natively as possible.

The *Solid.vertices()* and *Solid.wcs_vertices()* methods return the vertices in the *expected* (reversed) order.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'SOLID'
Factory function	*ezdxf.layouts.BaseLayout.add_solid()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Solid**

dxf.**vtx0**

Location of 1. vertex (2D/3D Point in *OCS*)

`dxf.`**`vtx1`**

Location of 2. vertex (2D/3D Point in *OCS*)

`dxf.`**`vtx2`**

Location of 3. vertex (2D/3D Point in *OCS*)

`dxf.`**`vtx3`**

Location of 4. vertex (2D/3D Point in *OCS*)

`transform`(*m:* Matrix44) → Solid

Transform the SOLID/TRACE entity by transformation matrix *m* inplace.

`vertices`(*close: bool = False*) → list[*ezdxf.math._vector.Vec3*]

Returns OCS vertices in correct order, if argument *close* is `True`, last vertex == first vertex. Does **not** return the duplicated last vertex if the entity represents a triangle.

`wcs_vertices`(*close: bool = False*) → list[*ezdxf.math._vector.Vec3*]

Returns WCS vertices in correct order, if argument *close* is `True`, last vertex == first vertex. Does **not** return the duplicated last vertex if the entity represents a triangle.

Spline

The SPLINE entity (DXF Reference) is a 3D curve, all coordinates have to be 3D coordinates even if the spline is just a 2D planar curve.

The spline curve is defined by control points, knot values and weights. The control points establish the spline, the various types of knot vector determines the shape of the curve and the weights of rational splines define how strong a control point influences the shape.

A SPLINE can be created just from fit points - knot values and weights are optional (tested with AutoCAD 2010). If you add additional data, be sure you know what you do, because invalid data may invalidate the whole DXF file.

The function *`ezdxf.math.fit_points_to_cad_cv()`* calculates control vertices from given fit points. This control vertices define a cubic B-spline which matches visually the SPLINE entities created by BricsCAD and AutoCAD from fit points.

See also:

- Wikipedia article about B_splines
- Department of Computer Science and Technology at the Cambridge University
- *Tutorial for Spline*

Subclass of	*`ezdxf.entities.DXFGraphic`*
DXF type	`'SPLINE'`
Factory function	see table below
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

Factory Functions

Basic spline entity	*add_spline()*
Spline control frame from fit points	*add_spline_control_frame()*
Open uniform spline	*add_open_spline()*
Closed uniform spline	add_closed_spline()
Open rational uniform spline	*add_rational_spline()*
Closed rational uniform spline	add_closed_rational_spline()

class ezdxf.entities.**Spline**

All points in *WCS* as (x, y, z) tuples

dxf.**degree**

Degree of the spline curve (int).

dxf.**flags**

Bit coded option flags, constants defined in *ezdxf.lldxf.const*:

dxf.flags	Value	Description
CLOSED_SPLINE	1	Spline is closed
PERIODIC_SPLINE	2	
RATIONAL_SPLINE	4	
PLANAR_SPLINE	8	
LINEAR_SPLINE	16	planar bit is also set

dxf.**n_knots**

Count of knot values (int), automatically set by *ezdxf* (read only)

dxf.**n_fit_points**

Count of fit points (int), automatically set by ezdxf (read only)

dxf.**n_control_points**

Count of control points (int), automatically set by ezdxf (read only)

dxf.**knot_tolerance**

Knot tolerance (float); default is 1e-10

dxf.**fit_tolerance**

Fit tolerance (float); default is 1e-10

dxf.**control_point_tolerance**

Control point tolerance (float); default is 1e-10

dxf.**start_tangent**

Start tangent vector as 3D vector in *WCS*

dxf.**end_tangent**

End tangent vector as 3D vector in *WCS*

closed

True if spline is closed. A closed spline has a connection from the last control point to the first control point. (read/write)

control_points

VertexArray of control points in *WCS*.

fit_points

VertexArray of fit points in *WCS*.

knots

Knot values as `array.array('d')`.

weights

Control point weights as `array.array('d')`.

control_point_count() → int

Count of control points.

fit_point_count() → int

Count of fit points.

knot_count() → int

Count of knot values.

construction_tool() → *BSpline*

Returns the construction tool *ezdxf.math.BSpline*.

apply_construction_tool(*s*) → Spline

Apply SPLINE data from a *BSpline* construction tool or from a `geomdl.BSpline.Curve` object.

flattening(*distance: float*, *segments: int = 4*) → Iterator[*Vec3*]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments between two knots, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the projected curve point onto the segment chord.
- **segments** – minimum segment count between two knots

set_open_uniform(*control_points: Sequence[*UVec*]*, *degree: int = 3*) → None

Open B-spline with a uniform knot vector, start and end at your first and last control points.

set_uniform(*control_points: Sequence[*UVec*]*, *degree: int = 3*) → None

B-spline with a uniform knot vector, does NOT start and end at your first and last control points.

set_closed(*control_points: Sequence[*UVec*]*, *degree=3*) → None

Closed B-spline with a uniform knot vector, start and end at your first control point.

set_open_rational(*control_points: Sequence[*UVec*]*, *weights: Sequence[float]*, *degree: int = 3*) → None

Open rational B-spline with a uniform knot vector, start and end at your first and last control points, and has additional control possibilities by weighting each control point.

set_uniform_rational(*control_points: Sequence[*UVec*]*, *weights: Sequence[float]*, *degree: int = 3*) → None

Rational B-spline with a uniform knot vector, does NOT start and end at your first and last control points, and has additional control possibilities by weighting each control point.

set_closed_rational(*control_points: Sequence[UVec]*, *weights: Sequence[float]*, *degree: int = 3*) → None

Closed rational B-spline with a uniform knot vector, start and end at your first control point, and has additional control possibilities by weighting each control point.

transform(*m:* Matrix44) → Spline

Transform the SPLINE entity by transformation matrix *m* inplace.

classmethod from_arc(*entity: DXFGraphic*) → Spline

Create a new SPLINE entity from a CIRCLE, ARC or ELLIPSE entity.

The new SPLINE entity has no owner, no handle, is not stored in the entity database nor assigned to any layout!

Surface

SURFACE entity (DXF Reference) created by an ACIS geometry kernel provided by the Spatial Corp.

See also:

Ezdxf has only very limited support for ACIS based entities, for more information see the FAQ: *How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?*

Subclass of	*ezdxf.entities.Body*
DXF type	'SURFACE'
Factory function	*ezdxf.layouts.BaseLayout.add_surface()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Surface**

Same attributes and methods as parent class *Body*.

dxf.**u_count**

Number of U isolines.

dxf.**v_count**

Number of V2 isolines.

ExtrudedSurface

(DXF Reference)

Subclass of	*ezdxf.entities.Surface*
DXF type	'EXTRUDEDSURFACE'
Factory function	*ezdxf.layouts.BaseLayout.add_extruded_surface()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2007 ('AC1021')

class ezdxf.entities.**ExtrudedSurface**

Same attributes and methods as parent class *Surface*.

dxf.**class_id**

dxf.**sweep_vector**

dxf.**draft_angle**

dxf.**draft_start_distance**

dxf.**draft_end_distance**

dxf.**twist_angle**

dxf.**scale_factor**

dxf.**align_angle**

dxf.**solid**

dxf.**sweep_alignment_flags**

0	No alignment
1	Align sweep entity to path
2	Translate sweep entity to path
3	Translate path to sweep entity

dxf.**align_start**

dxf.**bank**

dxf.**base_point_set**

dxf.**sweep_entity_transform_computed**

dxf.**path_entity_transform_computed**

dxf.**reference_vector_for_controlling_twist**

transformation_matrix_extruded_entity

type: *Matrix44*

sweep_entity_transformation_matrix

type: *Matrix44*

path_entity_transformation_matrix

type: *Matrix44*

LoftedSurface

(DXF Reference)

Subclass of	*ezdxf.entities.Surface*
DXF type	'LOFTEDSURFACE'
Factory function	*ezdxf.layouts.BaseLayout.add_lofted_surface()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2007 ('AC1021')

class ezdxf.entities.**LoftedSurface**

Same attributes and methods as parent class *Surface*.

dxf.**plane_normal_lofting_type**

dxf.**start_draft_angle**

dxf.**end_draft_angle**

dxf.**start_draft_magnitude**

dxf.**end_draft_magnitude**

dxf.**arc_length_parameterization**

dxf.**no_twist**

dxf.**align_direction**

dxf.**simple_surfaces**

dxf.**closed_surfaces**

dxf.**solid**

dxf.**ruled_surface**

dxf.**virtual_guide**

set_transformation_matrix_lofted_entity

type: *Matrix44*

RevolvedSurface

(DXF Reference)

Subclass of	*ezdxf.entities.Surface*
DXF type	'REVOLVEDSURFACE'
Factory function	*ezdxf.layouts.BaseLayout.add_revolved_surface()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2007 ('AC1021')

class ezdxf.entities.**RevolvedSurface**

Same attributes and methods as parent class *Surface*.

dxf.**class_id**

dxf.**axis_point**

dxf.**axis_vector**

dxf.**revolve_angle**

dxf.**start_angle**

dxf.**draft_angle**

dxf.**start_draft_distance**

dxf.**end_draft_distance**

dxf.**twist_angle**

dxf.**solid**

dxf.**close_to_axis**

transformation_matrix_revolved_entity

type: *Matrix44*

SweptSurface

(DXF Reference)

Subclass of	*ezdxf.entities.Surface*
DXF type	'SWEPTSURFACE'
Factory function	*ezdxf.layouts.BaseLayout.add_swept_surface()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2007 ('AC1021')

class ezdxf.entities.**SweptSurface**

Same attributes and methods as parent class *Surface*.

dxf.**swept_entity_id**

dxf.**path_entity_id**

dxf.**draft_angle**

draft_start_distance

dxf.**draft_end_distance**

dxf.**twist_angle**

dxf.**scale_factor**

dxf.**align_angle**

dxf.**solid**

dxf.**sweep_alignment**

dxf.**align_start**

dxf.**bank**

dxf.**base_point_set**

dxf.**sweep_entity_transform_computed**

dxf.**path_entity_transform_computed**

dxf.**reference_vector_for_controlling_twist**

transformation_matrix_sweep_entity
type: *Matrix44*

transformation_matrix_path_entity()
type: *Matrix44*

sweep_entity_transformation_matrix()
type: *Matrix44*

path_entity_transformation_matrix()
type: *Matrix44*

Text

The TEXT entity (DXF Reference) represents a single line of text. The *style* attribute stores the associated *Textstyle* entity as string, which defines the basic font properties. The text size is stored as cap-height in the *height* attribute in drawing units. Text alignments are defined as enums of type *ezdxf.enums.TextEntityAlignment*.

See also:

See the documentation for the *Textstyle* class to understand the limitations of text representation in the DXF format.

Tutorial for Text

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'TEXT'
Factory function	*ezdxf.layouts.BaseLayout.add_text()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Text**

dxf.**text**
Text content as string.

dxf.**insert**
First alignment point of text (2D/3D Point in *OCS*), relevant for the adjustments LEFT, ALIGNED and FIT.

dxf.**align_point**

The main alignment point of text (2D/3D Point in *OCS*), if the alignment is anything else than LEFT, or the second alignment point for the ALIGNED and FIT alignments.

dxf.**height**

Text height in drawing units as float value, the default value is 1.

dxf.**rotation**

Text rotation in degrees as float value, the default value is 0.

dxf.**oblique**

Text oblique angle (slanting) in degrees as float value, the default value is 0 (straight vertical text).

dxf.**style**

Textstyle name as case insensitive string, the default value is "Standard"

dxf.**width**

Width scale factor as float value, the default value is 1.

dxf.**halign**

Horizontal alignment flag as int value, use the *set_placement()* and *get_align_enum()* methods to handle text alignment, the default value is 0.

0	Left
2	Right
3	Aligned (if vertical alignment = 0)
4	Middle (if vertical alignment = 0)
5	Fit (if vertical alignment = 0)

dxf.**valign**

Vertical alignment flag as int value, use the *set_placement()* and *get_align_enum()* methods to handle text alignment, the default value is 0.

0	Baseline
1	Bottom
2	Middle
3	Top

dxf.**text_generation_flag**

Text generation flags as int value, use the `is_backward` and `is_upside_down` attributes to handle this flags.

2	text is backward (mirrored in X)
4	text is upside down (mirrored in Y)

property is_backward: bool

Get/set text generation flag BACKWARDS, for mirrored text along the x-axis.

property is_upside_down: bool

Get/set text generation flag UPSIDE_DOWN, for mirrored text along the y-axis.

set_placement(*p1:* UVec, *p2:* UVec | *None* = *None*, *align:* TextEntityAlignment | *None* = *None*) → Text

Set text alignment and location.

The alignments `ALIGNED` and `FIT` are special, they require a second alignment point, the text is aligned on the virtual line between these two points and sits vertically at the baseline.

- `ALIGNED`: Text is stretched or compressed to fit exactly between *p1* and *p2* and the text height is also adjusted to preserve height/width ratio.
- `FIT`: Text is stretched or compressed to fit exactly between *p1* and *p2* but only the text width is adjusted, the text height is fixed by the *dxf.height* attribute.
- `MIDDLE`: also a special adjustment, centered text like `MIDDLE_CENTER`, but vertically centred at the total height of the text.

Parameters

- **p1** – first alignment point as (x, y[, z])
- **p2** – second alignment point as (x, y[, z]), required for `ALIGNED` and `FIT` else ignored
- **align** – new alignment as enum *TextEntityAlignment*, `None` to preserve the existing alignment.

get_placement() → tuple[*ezdxf.enums.TextEntityAlignment*, *ezdxf.math._vector.Vec3*, Optional[*ezdxf.math._vector.Vec3*]]

Returns a tuple (*align*, *p1*, *p2*), *align* is the alignment enum `TextEntityAlignment`, *p1* is the alignment point, *p2* is only relevant if *align* is `ALIGNED` or `FIT`, otherwise it is `None`.

get_align_enum() → *TextEntityAlignment*

Returns the current text alignment as *TextEntityAlignment*, see also *set_placement()*.

set_align_enum(*align=TextEntityAlignment.LEFT*) → Text

Just for experts: Sets the text alignment without setting the alignment points, set adjustment points attr:*dxf.insert* and *dxf.align_point* manually.

Parameters

align – *TextEntityAlignment*

transform(*m:* Matrix44) → Text

Transform the TEXT entity by transformation matrix *m* inplace.

translate(*dx: float*, *dy: float*, *dz: float*) → Text

Optimized TEXT/ATTRIB/ATTDEF translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self*.

plain_text() → str

Returns text content without formatting codes.

font_name() → str

Returns the font name of the associated *Textstyle*.

fit_length() → float

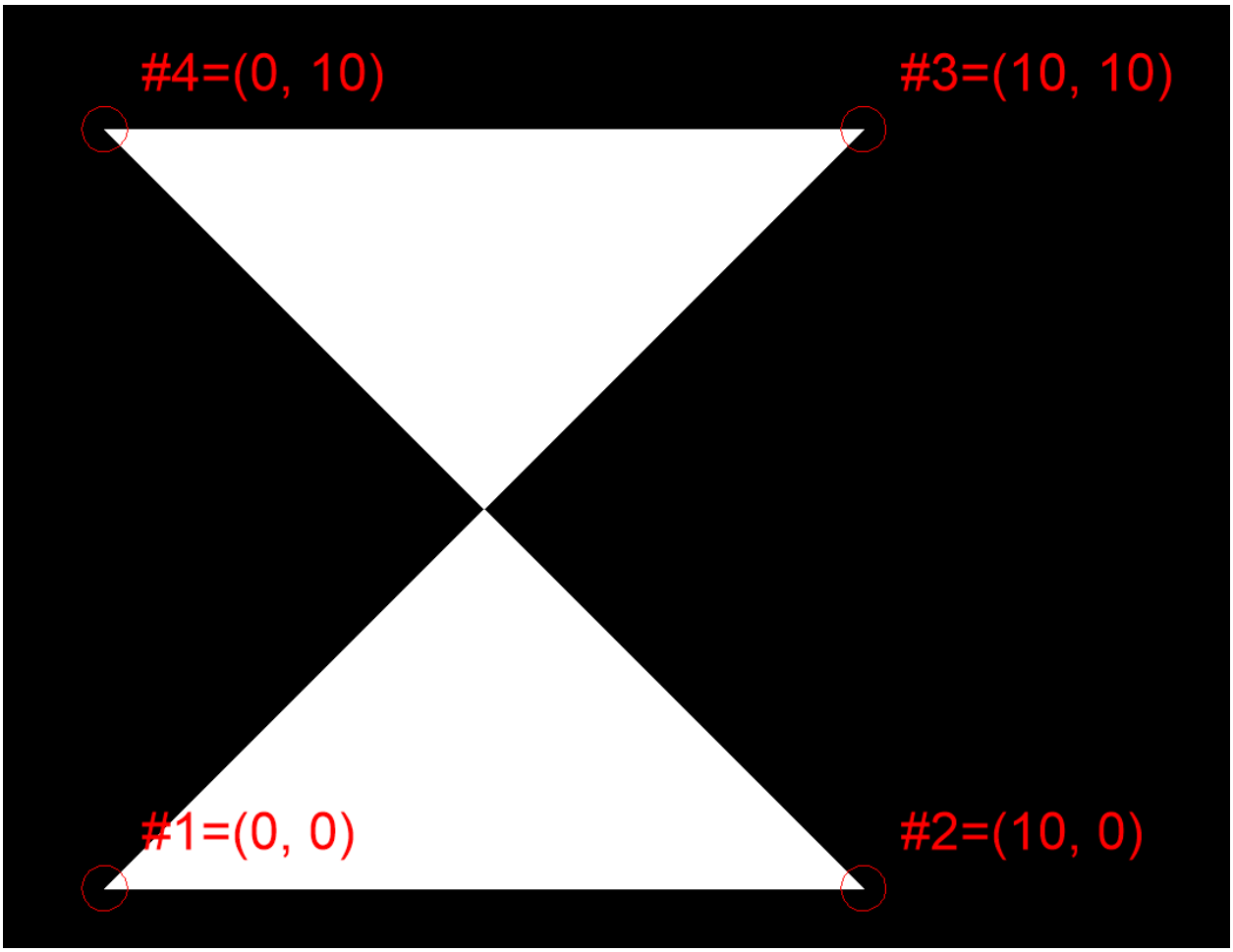
Returns the text length for alignments `TextEntityAlignment.FIT` and `TextEntityAlignment.ALIGNED`, defined by the distance from the insertion point to the align point or 0 for all other alignments.

Trace

The TRACE entity (DXF Reference) is solid filled triangle or quadrilateral. Access vertices by name (`entity.dxf.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`). If only 3 vertices are provided the last (3rd) vertex will be repeated in the DXF file.

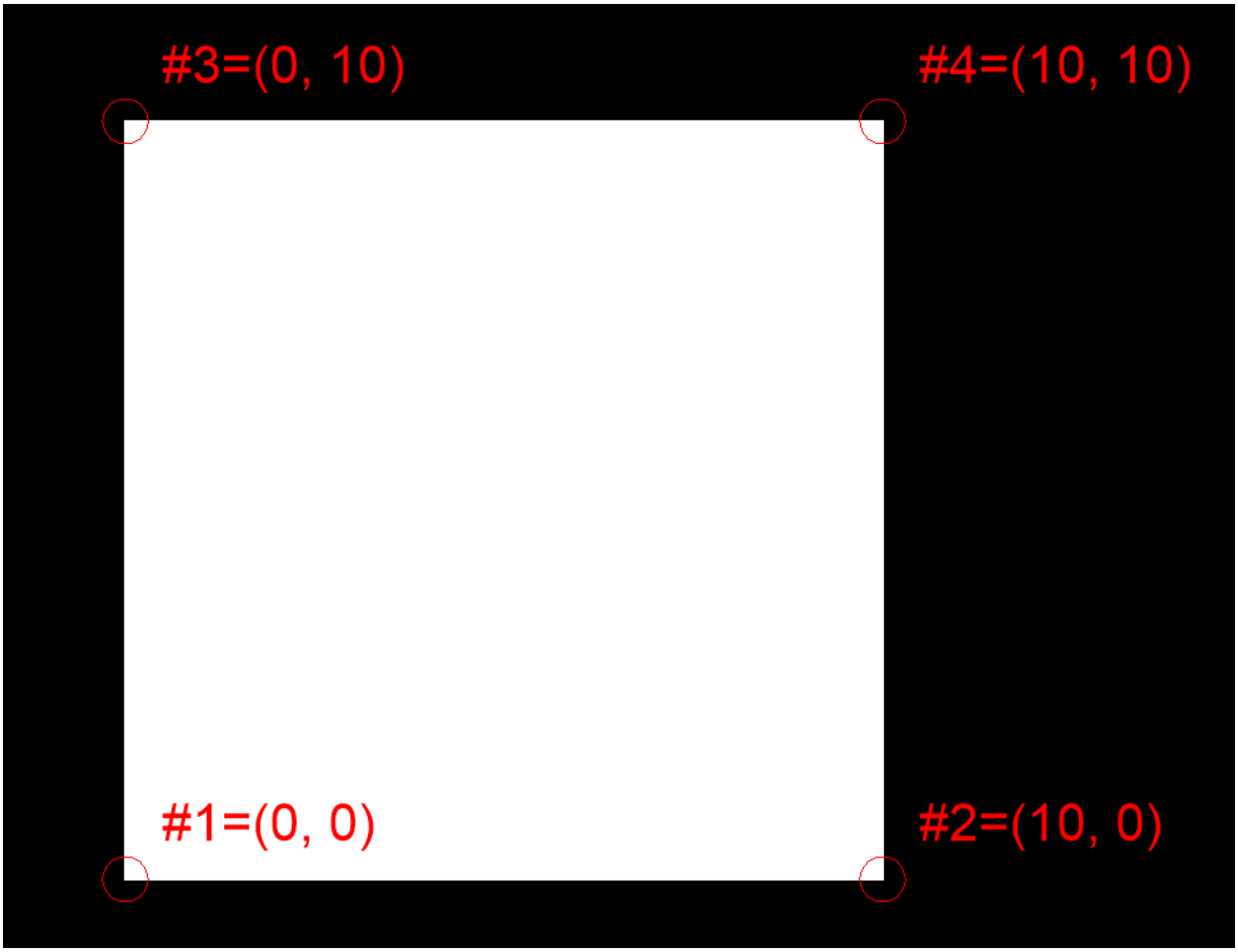
The TRACE entity stores the vertices in an unusual way, the last two vertices are reversed:

```
msp.add_solid([(0, 0), (10, 0), (10, 10), (0, 10)])
```



Reverse the last two vertices to get the *expected* square:

```
msp.add_solid([(0, 0), (10, 0), (0, 10), (10, 10)])
```



Note: The quirky vertex order is preserved at the lowest access level because *ezdxf* is intended as a DXF file format interface and presents the content of the DXF document to the package user as natively as possible.

The *Trace.vertices()* and *Trace.wcs_vertices()* methods return the vertices in the *expected* (reversed) order.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'TRACE'
Factory function	*ezdxf.layouts.BaseLayout.add_trace()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Trace**

dxf.**vtx0**
Location of 1. vertex (2D/3D Point in *OCS*)

dxf.**vtx1**
Location of 2. vertex (2D/3D Point in *OCS*)

dxf.**vtx2**

Location of 3. vertex (2D/3D Point in *OCS*)

dxf.**vtx3**

Location of 4. vertex (2D/3D Point in *OCS*)

transform (*m:* Matrix44) → Solid

Transform the SOLID/TRACE entity by transformation matrix *m* inplace.

vertices (*close: bool = False*) → list[*ezdxf.math._vector.Vec3*]

Returns OCS vertices in correct order, if argument *close* is `True`, last vertex == first vertex. Does **not** return the duplicated last vertex if the entity represents a triangle.

wcs_vertices (*close: bool = False*) → list[*ezdxf.math._vector.Vec3*]

Returns WCS vertices in correct order, if argument *close* is `True`, last vertex == first vertex. Does **not** return the duplicated last vertex if the entity represents a triangle.

Underlay

The UNDERLAY entity (DXF Reference) links an underlay file to the DXF file, the file itself is not embedded into the DXF file, it is always a separated file. The (PDF)UNDERLAY entity is like a block reference, you can use it multiple times to add the underlay on different locations with different scales and rotations. But therefore you need a also a (PDF)DEFINITION entity, see *UnderlayDefinition*.

The DXF standard supports three different file formats: PDF, DWF (DWFx) and DGN. An Underlay can be clipped by a rectangle or a polygon path. The clipping coordinates are 2D *OCS* coordinates in drawing units but without scaling.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	internal base class
Factory function	*ezdxf.layouts.BaseLayout.add_underlay()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 (`'AC1015'`)

class ezdxf.entities.**Underlay**

Base class of *PdfUnderlay*, *DwfUnderlay* and *DgnUnderlay*

dxf.**insert**

Insertion point, lower left corner of the image in *OCS*.

dxf.**scale_x**

Scaling factor in x-direction (float)

dxf.**scale_y**

Scaling factor in y-direction (float)

dxf.**scale_z**

Scaling factor in z-direction (float)

dxf.**rotation**

ccw rotation in degrees around the extrusion vector (float)

dxf.**extrusion**

extrusion vector, default is (0, 0, 1)

`dxf.`**underlay_def_handle**

Handle to the underlay definition entity, see *UnderlayDefinition*

`dxf.`**flags**

dxf.flags	Value	Description
UNDERLAY_CLIPPING	1	clipping is on/off
UNDERLAY_ON	2	underlay is on/off
UNDERLAY_MONOCHROME	4	Monochrome
UNDERLAY_ADJUST_FOR_BACKGROUND	8	Adjust for background

`dxf.`**contrast**

Contrast value (20 - 100; default is 100)

`dxf.`**fade**

Fade value (0 - 80; default is 0)

clipping

`True` or `False` (read/write)

on

`True` or `False` (read/write)

monochrome

`True` or `False` (read/write)

adjust_for_background

`True` or `False` (read/write)

scale

Scaling (x, y, z) tuple (read/write)

boundary_path

Boundary path as list of vertices (read/write).

Two vertices describe a rectangle (lower left and upper right corner), more than two vertices is a polygon as clipping path.

get_underlay_def()

Returns the associated DEFINITION entity. see *UnderlayDefinition*.

set_underlay_def()

Set the associated DEFINITION entity. see *UnderlayDefinition*.

reset_boundary_path() → None

Removes the clipping path.

PdfUnderlay

Subclass of	*ezdxf.entities.Underlay*
DXF type	'PDFUNDERLAY'
Factory function	*ezdxf.layouts.BaseLayout.add_underlay()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**PdfUnderlay**

PDF underlay.

DwfUnderlay

Subclass of	*ezdxf.entities.Underlay*
DXF type	'DWFUNDERLAY'
Factory function	*ezdxf.layouts.BaseLayout.add_underlay()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**DwfUnderlay**

DWF underlay.

DgnUnderlay

Subclass of	*ezdxf.entities.Underlay*
DXF type	'DGNUNDERLAY'
Factory function	*ezdxf.layouts.BaseLayout.add_underlay()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**DgnUnderlay**

DGN underlay.

Viewport

The VIEWPORT entity (DXF Reference) is a window from a paperspace layout to the modelspace.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'VIEWPORT'
Factory function	*ezdxf.layouts.Paperspace.add_viewport()*
Inherited DXF attributes	*Common graphical DXF attributes*

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Viewport**

dxf.**center**

Center point of the viewport located in the paper space layout in paper space units stored as 3D point. (Error in the DXF reference)

dxf.**width**

Viewport width in paperspace units (float)

dxf.**height**

Viewport height in paperspace units (float)

dxf.**status**

Viewport status field (int)

-1	On, but is fully off screen, or is one of the viewports that is not active because the $MAXACTVP count is currently being exceeded.
0	Off
>0	On and active. The value indicates the order of stacking for the viewports, where 1 is the active viewport, 2 is the next, and so forth

dxf.**id**

Viewport id (int)

dxf.**view_center_point**

View center point in modelspace stored as 2D point, but represents a *WCS* point. (Error in the DXF reference)

dxf.**snap_base_point**

dxf.**snap_spacing**

dxf.**snap_angle**

dxf.**grid_spacing**

dxf.**view_direction_vector**

View direction (3D vector in *WCS*).

dxf.**view_target_point**

View target point (3D point in *WCS*).

dxf.**perspective_lens_length**

Lens focal length in mm as 35mm film equivalent.

dxf.**front_clip_plane_z_value**

dxf.**back_clip_plane_z_value**

dxf.**view_height**

View height in *WCS*.

dxf.**view_twist_angle**

dxf.**circle_zoom**

dxf.**flags**

Viewport status bit-coded flags:

Bit value	Constant in ezdxf.const	Description
1 (0x1)	VSF_PERSPECTIVE	Enables perspective mode
2 (0x2)	VSF_FRONT_CLIPP	Enables front clipping
4 (0x4)	VSF_BACK_CLIPPI	Enables back clipping
8 (0x8)	VSF_USC_FOLLOW	Enables UCS follow
16 (0x10)	VSF_FRONT_CLIPP	Enables front clip not at eye
32 (0x20)	VSF_UCS_ICON_VI	Enables UCS icon visibility
64 (0x40)	VSF_UCS_ICON_AT	Enables UCS icon at origin
128 (0x80)	VSF_FAST_ZOOM	Enables fast zoom
256 (0x100)	VSF_SNAP_MODE	Enables snap mode
512 (0x200)	VSF_GRID_MODE	Enables grid mode
1024 (0x400)	VSF_ISOMETRIC_S	Enables isometric snap style
2048 (0x800)	VSF_HIDE_PLOT_M	Enables hide plot mode
4096 (0x1000)	VSF_KISOPAIR_TO	kIsoPairTop. If set and kIsoPairRight is not set, then isopair top is enabled. If both kIsoPairTop and kIsoPairRight are set, then isopair left is enabled
8192 (0x2000)	VSF_KISOPAIR_RIC	kIsoPairRight. If set and kIsoPairTop is not set, then isopair right is enabled
16384 (0x4000)	VSF_LOCK_ZOOM	Enables viewport zoom locking
32768 (0x8000)	VSF_CURRENTLY_	Currently always enabled
65536 (0x10000)	VSF_NON_RECTAN	Enables non-rectangular clipping
131072 (0x20000)	VSF_TURN_VIEWP	Turns the viewport off
262144 (0x40000)	VSF_NO_GRID_LIM	Enables the display of the grid beyond the drawing limits
524288 (0x80000)	VSF_ADAPTIVE_GI	Enable adaptive grid display
1048576 (0x100000)	VSF_SUBDIVIDE_G	Enables subdivision of the grid below the set grid spacing when the grid display is adaptive
2097152 (0x200000)	VSF_GRID_FOLLOW	Enables grid follows workplane switching

Use helper method *set_flag_state()* to set and clear viewport flags, e.g. lock viewport:

```
vp.set_flag_state(ezdxf.const.VSF_LOCK_ZOOM, True)
```

dxf.**clipping_boundary_handle**

dxf.**plot_style_name**

dxf.**render_mode**

0	2D Optimized (classic 2D)
1	Wireframe
2	Hidden line
3	Flat shaded
4	Gouraud shaded
5	Flat shaded with wireframe
6	Gouraud shaded with wireframe

dxf.**ucs_per_viewport**

dxf.**ucs_icon**

dxf.**ucs_origin**
UCS origin as 3D point.

dxf.**ucs_x_axis**
UCS x-axis as 3D vector.

dxf.**ucs_y_axis**
UCS y-axis as 3D vector.

dxf.**ucs_handle**
Handle of `UCSTable` if UCS is a named UCS. If not present, then UCS is unnamed.

dxf.**ucs_ortho_type**

0	not orthographic
1	Top
2	Bottom
3	Front
4	Back
5	Left
6	Right

dxf.**ucs_base_handle**
Handle of `UCSTable` of base UCS if UCS is orthographic (*Viewport.dxf.ucs_ortho_type* is non-zero). If not present and *Viewport.dxf.ucs_ortho_type* is non-zero, then base UCS is taken to be WORLD.

dxf.**elevation**

dxf.**shade_plot_mode**
(DXF R2004)

0	As Displayed
1	Wireframe
2	Hidden
3	Rendered

dxf.**grid_frequency**
Frequency of major grid lines compared to minor grid lines. (DXF R2007)

dxf.**background_handle**

dxf.**shade_plot_handle**

dxf.**visual_style_handle**

dxf.**default_lighting_flag**

dxf.**default_lighting_style**

0	One distant light
1	Two distant lights

dxf.**view_brightness**

dxf.**view_contrast**

dxf.**ambient_light_color_1**
as *AutoCAD Color Index (ACI)*

dxf.**ambient_light_color_2**
as true color value

dxf.**ambient_light_color_3**
as true color value

dxf.**sun_handle**

dxf.**ref_vp_object_1**

dxf.**ref_vp_object_2**

dxf.**ref_vp_object_3**

dxf.**ref_vp_object_4**

frozen_layers
Set/get frozen layers as list of layer names.

is_frozen(*layer_name: str*) → bool
Returns `True` if *layer_name* id frozen in this viewport.

freeze(*layer_name: str*) → None
Freeze *layer_name* in this viewport.

thaw(*layer_name: str*) → None
Thaw *layer_name* in this viewport.

has_extended_clipping_path
Returns `True` if a non-rectangular clipping path is defined.

clipping_rect() → tuple[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec2*]
Returns the lower left and the upper right corner of the clipping rectangle.

clipping_rect_corners() → list[*ezdxf.math._vector.Vec2*]

Returns the default rectangular clipping path as list of vertices. Use function *ezdxf.path.make_path()* to get also non-rectangular shaped clipping paths if defined.

get_aspect_ratio() → float

Returns the aspect ratio of the viewport, return 0.0 if width or height is zero.

get_modelspace_limits() → tuple[float, float, float, float]

Returns the limits of the modelspace to view in drawing units as tuple (min_x, min_y, max_x, max_y).

get_scale() → float

Returns the scaling factor from modelspace to viewport.

get_transformation_matrix() → *Matrix44*

Returns the transformation matrix from modelspace to viewport.

Wipeout

The WIPEOUT entity (DXF Reference) is a polygonal area that masks underlying objects with the current background color. The WIPEOUT entity is based on the IMAGE entity, but usage does not require any knowledge about the IMAGE entity.

The handles to the support entities *ImageDef* and *ImageDefReactor* are always "0", both are not needed by the WIPEOUT entity.

Subclass of	*ezdxf.entities.Image*
DXF type	'WIPEOUT'
Factory function	*ezdxf.layouts.BaseLayout.add_wipeout()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Wipeout**

set_masking_area(*vertices: Iterable[*UVec*]*) → None

Set a new masking area, the area is placed in the layout xy-plane.

XLine

The XLINE entity (DXF Reference) is a construction line that extents to infinity in both directions.

Subclass of	*ezdxf.entities.DXFGraphic*
DXF type	'XLINE'
Factory function	*ezdxf.layouts.BaseLayout.add_xline()*
Inherited DXF attributes	*Common graphical DXF attributes*
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.**XLine**

`dxf.start`

Location point of line as (3D Point in *WCS*)

`dxf.unit_vector`

Unit direction vector as (3D Point in *WCS*)

transform(*m:* Matrix44) → XLine

Transform the XLINE/RAY entity by transformation matrix *m* inplace.

translate(*dx: float*, *dy: float*, *dz: float*) → XLine

Optimized XLINE/RAY translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

DXF Objects

All DXF objects can only reside in the OBJECTS section of a DXF document.

The purpose of the OBJECTS section is to allow CAD software developers to define and store custom objects that are not included in the basic DXF file format. These custom objects can be used to represent complex data structures, such as database tables or project management information, that are not easily represented by basic DXF entities.

By including custom objects in the OBJECTS section, CAD software developers can extend the functionality of their software to support new types of data and objects. For example, a custom application might define a new type of block or dimension style that is specific to a particular industry or workflow. By storing this custom object definition in the OBJECTS section, the CAD software can recognize and use the new object type in a drawing.

In summary, the OBJECTS section is an important part of the DXF file format because it allows CAD software developers to extend the functionality of their software by defining and storing custom objects and entity types. This makes it possible to represent complex data structures and workflows in CAD drawings, and allows CAD software to be customized to meet the specific needs of different industries and applications.

Dictionary

The DICTIONARY entity is a general storage entity.

AutoCAD maintains items such as MLINE_STYLES and GROUP definitions as objects in dictionaries. Other applications are free to create and use their own dictionaries as they see fit. The prefix `'ACAD_'` is reserved for use by AutoCAD applications.

Dictionary entries are (key, *DXFEntity*) pairs for fully loaded or new created DXF documents. The referenced entities are owned by the dictionary and cannot be graphical entities that always belong to the layout in which they are located.

Loading DXF files is done in two passes, because at the first loading stage not all referenced objects are already stored in the entity database. Therefore the entities are stored as handles strings at the first loading stage and have to be replaced by the real entity at the second loading stage. If the entity is still a handle string after the second loading stage, the entity does not exist.

Dictionary keys are handled case insensitive by AutoCAD, but not by *ezdxf*, in doubt use an uppercase key. AutoCAD stores all keys in uppercase.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	`'DICTIONARY'`
Factory function	*ezdxf.sections.objects.ObjectsSection.add_dictionary()*

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Dictionary**

dxf.**hard_owned**

If set to 1, indicates that elements of the dictionary are to be treated as hard-owned.

dxf.**cloning**

Duplicate record cloning flag (determines how to merge duplicate entries, ignored by *ezdxf*):

0	not applicable
1	keep existing
2	use clone
3	<xref>0<name>
4	0<name>
5	Unmangle name

is_hard_owner

Returns `True` if the dictionary is hard owner of entities. Hard owned entities will be destroyed by deleting the dictionary.

__len__() → int

Returns count of dictionary entries.

__contains__(*key: str*) → bool

Returns *key* `in` self.

__getitem__(*key: str*) → DXFEntity

Return self[*key*].

The returned value can be a handle string if the entity does not exist.

Raises

DXFKeyError – *key* does not exist

__setitem__(*key: str*, *entity: DXFObject*) → None

Set self[*key*] = *entity*.

Only DXF objects stored in the OBJECTS section are allowed as content of *Dictionary* objects. DXF entities stored in layouts are not allowed.

Raises

DXFTypeError – invalid DXF type

__delitem__(*key: str*) → None

Delete self[*key*].

Raises

DXFKeyError – *key* does not exist

keys()

Returns a `KeysView` of all dictionary keys.

items()

Returns an `ItemsView` for all dictionary entries as (key, entity) pairs. An entity can be a handle string if the entity does not exist.

count() → int

Returns count of dictionary entries.

get(*key: str*, *default: DXFObject | None = None*) → DXFObject | None

Returns the *DXFEntity* for *key*, if *key* exist else *default*. An entity can be a handle string if the entity does not exist.

add(*key: str*, *entity: DXFObject*) → None

Add entry (key, value).

If the DICTIONARY is hard owner of its entries, the *add()* does NOT take ownership of the entity automatically.

Raises

- ***DXFValueError*** – invalid entity handle
- ***DXFTypeError*** – invalid DXF type

remove(*key: str*) → None

Delete entry *key*. Raises `DXFKeyError`, if *key* does not exist. Destroys hard owned DXF entities.

discard(*key: str*) → None

Delete entry *key* if exists. Does not raise an exception if *key* doesn't exist and does not destroy hard owned DXF entities.

clear() → None

Delete all entries from the dictionary and destroys hard owned DXF entities.

add_new_dict(*key: str*, *hard_owned: bool = False*) → Dictionary

Create a new sub-dictionary of type *Dictionary*.

Parameters

- **key** – name of the sub-dictionary
- **hard_owned** – entries of the new dictionary are hard owned

get_required_dict(*key: str*, *hard_owned=False*) → Dictionary

Get entry *key* or create a new *Dictionary*, if *Key* not exist.

add_dict_var(*key: str*, *value: str*) → DictionaryVar

Add a new *DictionaryVar*.

Parameters

- **key** – entry name as string
- **value** – entry value as string

add_xrecord(*key: str*) → *XRecord*

Add a new *XRecord*.

Parameters
key – entry name as string

link_dxf_object(*name: str*, *obj: DXFObject*) → None

Add *obj* and set owner of *obj* to this dictionary.

Graphical DXF entities have to reside in a layout and therefore can not be owned by a *Dictionary*.

Raises
DXFTypeError – *obj* has invalid DXF type

DictionaryWithDefault

Subclass of	*ezdxf.entities.Dictionary*
DXF type	'ACDBDICTIONARYWDFLT'
Factory function	*ezdxf.sections.objects.ObjectsSection.add_dictionary_with_default()*

class ezdxf.entities.**DictionaryWithDefault**

dxf.**default**

Handle to default entry as hex string like FF00.

get(*key: str*, *default: DXFObject | None = None*) → DXFObject | None

Returns *DXFEntity* for *key* or the predefined dictionary wide *dxf.default* entity if *key* does not exist or None if default value also not exist.

set_default(*default: DXFObject*) → None

Set dictionary wide default entry.

Parameters
default – default entry as *DXFEntity*

DictionaryVar

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'DICTIONARYVAR'
Factory function	*ezdxf.entities.Dictionary.add_dict_var()*

class ezdxf.entities.**DictionaryVar**

dxf.**schema**

Object schema number (currently set to 0)

dxf.**value**

Value as string.

property value: str

Get/set the value of the *DictionaryVar* as string.

DXFLayout

LAYOUT entity is part of a modelspace or paperspace layout definitions.

Subclass of	*ezdxf.entities.PlotSettings*
DXF type	'LAYOUT'
Factory function	internal data structure, use *Layouts* to manage layout objects.

class ezdxf.entities.**DXFLayout**

dxf.**name**
Layout name as shown in tabs by *CAD* applications

dxf.**layout_flags**

1	Indicates the PSLTSCALE value for this layout when this layout is current
2	Indicates the LIMCHECK value for this layout when this layout is current

dxf.**tab_order**
default is 1

dxf.**limmin**
default is Vec2(0, 0)

dxf.**limmax**
default is Vec2(420, 297)

dxf.**insert_base**
default is Vec3(0, 0, 0)

dxf.**extmin**
default is Vec3(1e20, 1e20, 1e20)

dxf.**extmax**
default is Vec3(-1e20, -1e20, -1e20)

dxf.**elevation**
default is 0

dxf.**ucs_origin**
default is Vec3(0, 0, 0)

dxf.**ucs_xaxis**
default is Vec3(1, 0, 0)

dxf.**ucs_yaxis**
default is Vec3(0, 1, 0)

dxf.**ucs_type**

0	UCS is not orthographic
1	Top
2	Bottom
3	Front
4	Back
5	Left
6	Right

default is 1

dxf.**block_record_handle**

dxf.**viewport_handle**

dxf.**ucs_handle**

dxf.**base_ucs_handle**

DXFObject

Common base class for all non-graphical DXF objects.

class ezdxf.entities.**DXFObject**

A class hierarchy marker class and subclass of *ezdxf.entities.DXFEntity*

GeoData

The GEODATA entity is associated to the *Modelspace* object. The GEODATA entity is supported since the DXF version R2000, but was officially documented the first time in the DXF reference for version R2009.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'GEODATA'
Factory function	*ezdxf.layouts.Modelspace.new_geodata()*
Required DXF version	R2010 ('AC1024')

See also:

geodata_setup_local_grid.py

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.**GeoData**

dxf.**version**

1	R2009
2	R2010

dxf.**coordinate_type**

0	unknown
1	local grid
2	projected grid
3	geographic (latitude/longitude)

dxf.**block_record_handle**

Handle of host BLOCK_RECORD table entry, in general the *Modelspace*.

dxf.**design_point**

Reference point in *WCS* coordinates.

dxf.**reference_point**

Reference point in geo coordinates, valid only when coordinate type is *local grid*. The difference between *dxf.design_point* and *dxf.reference_point* defines the translation from WCS coordinates to geo-coordinates.

dxf.**north_direction**

North direction as 2D vector. Defines the rotation (about the *dxf.design_point*) to transform from WCS coordinates to geo-coordinates

dxf.**horizontal_unit_scale**

Horizontal unit scale, factor which converts horizontal design coordinates to meters by multiplication.

dxf.**vertical_unit_scale**

Vertical unit scale, factor which converts vertical design coordinates to meters by multiplication.

dxf.**horizontal_units**

Horizontal units (see *BlockRecord*). Will be 0 (Unitless) if units specified by horizontal unit scale is not supported by AutoCAD enumeration.

dxf.**vertical_units**

Vertical units (see *BlockRecord*). Will be 0 (Unitless) if units specified by vertical unit scale is not supported by AutoCAD enumeration.

dxf.**up_direction**

Up direction as 3D vector.

dxf.**scale_estimation_method**

1	none
2	user specified scale factor
3	grid scale at reference point
4	prismoidal

dxf.**sea_level_correction**

Bool flag specifying whether to do sea level correction.

dxf.**user_scale_factor**

dxf.**sea_level_elevation**

dxf.**coordinate_projection_radius**

dxf.**geo_rss_tag**

dxf.**observation_from_tag**

dxf.**observation_to_tag**

dxf.**mesh_faces_count**

source_vertices

2D source vertices in the CRS of the GeoData as *VertexArray*. Used together with *target_vertices* to define the transformation from the CRS of the GeoData to WGS84.

target_vertices

2D target vertices in WGS84 (EPSG:4326) as *VertexArray*. Used together with *source_vertices* to define the transformation from the CRS of the geoData to WGS84.

faces

List of face definition tuples, each face entry is a 3-tuple of vertex indices (0-based).

coordinate_system_definition

The coordinate system definition string. Stored as XML. Defines the CRS used by the GeoData. The EPSG number and other details like the axis-ordering of the CRS is stored.

get_crs() → tuple[int, bool]

Returns the EPSG index and axis-ordering, axis-ordering is `True` if fist axis is labeled "E" or "W" and `False` if first axis is labeled "N" or "S".

If axis-ordering is `False` the CRS is not compatible with the `__geo_interface__` or GeoJSON (see chapter 3.1.1).

Raises

InvalidGeoDataException – for invalid or unknown XML data

The EPSG number is stored in a tag like:

```
<Alias id="27700" type="CoordinateSystem">
  <ObjectId>OSGB1936.NationalGrid</ObjectId>
  <Namespace>EPSG Code</Namespace>
</Alias>
```

The axis-ordering is stored in a tag like:

```
<Axis uom="METER">
  <CoordinateSystemAxis>
    <AxisOrder>1</AxisOrder>
    <AxisName>Easting</AxisName>
    <AxisAbbreviation>E</AxisAbbreviation>
    <AxisDirection>east</AxisDirection>
  </CoordinateSystemAxis>
  <CoordinateSystemAxis>
    <AxisOrder>2</AxisOrder>
    <AxisName>Northing</AxisName>
    <AxisAbbreviation>N</AxisAbbreviation>
    <AxisDirection>north</AxisDirection>
  </CoordinateSystemAxis>
</Axis>
```

get_crs_transformation(*, *no_checks: bool = False*) → tuple[*ezdxf.math._matrix44.Matrix44*, int]

Returns the transformation matrix and the EPSG index to transform WCS coordinates into CRS coordinates. Because of the lack of proper documentation this method works only for tested configurations, set argument *no_checks* to `True` to use the method for untested geodata configurations, but the results may be incorrect.

Supports only "Local Grid" transformation!

Raises

InvalidGeoDataException – for untested geodata configurations

setup_local_grid(*, *design_point:* UVec, *reference_point:* UVec, *north_direction:* UVec = *(0, 1)*, *crs: str = EPSG_3395*)

Setup local grid coordinate system. This method is designed to setup CRS similar to *EPSG:3395 World Mercator*, the basic features of the CRS should fulfill these assumptions:

- base unit of reference coordinates is 1 meter
- right-handed coordinate system: +Y=north/+X=east/+Z=up

The CRS string is not validated nor interpreted!

Hint: The reference point must be a 2D cartesian map coordinate and not a globe (lon/lat) coordinate like stored in GeoJSON or GPS data.

Parameters

- **design_point** – WCS coordinates of the CRS reference point
- **reference_point** – CRS reference point in 2D cartesian coordinates
- **north_direction** – north direction a 2D vertex, default is (0, 1)
- **crs** – Coordinate Reference System definition XML string, default is the definition string for *EPSG:3395 World Mercator*

ImageDef

The IMAGEDEF entity defines an image file, which can be placed by the *Image* entity.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'IMAGEDEF'
Factory function (1)	*ezdxf.document.Drawing.add_image_def()*
Factory function (2)	*ezdxf.sections.objects.ObjectsSection.add_image_def()*

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.**ImageDef**

dxf.**class_version**
Current version is 0.

dxf.**filename**
Relative (to the DXF file) or absolute path to the image file as string.

dxf.**image_size**
Image size in pixel as (x, y) tuple.

dxf.**pixel_size**
Default size of one pixel in drawing units as (x, y) tuple.

dxf.**loaded**
0 = unloaded; 1 = loaded, default is 1

dxf.**resolution_units**

0	No units
2	Centimeters
5	Inch

default is 0

ImageDefReactor

class ezdxf.entities.**ImageDefReactor**

dxf.**class_version**

dxf.**image_handle**

MLeaderStyle

The MLEADERSTYLE entity (DXF Reference) stores all attributes required to create new *MultiLeader* entities. The meaning of these attributes are not really documented in the DXF Reference. The default style "Standard" always exist.

See also:

- *ezdxf.entities.MultiLeader*
- *ezdxf.render.MultiLeaderBuilder*
- *Tutorial for MultiLeader*

Create a new *MLeaderStyle*:

```
import ezdxf

doc = ezdxf.new()
new_style = doc.mleader_styles.new("NewStyle")
```

Duplicate an existing style:

```
duplicated_style = doc.mleader_styles.duplicate_entry("Standard", "DuplicatedStyle")
```

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'MLEADERSTYLE'
Factory function	ezdxf.document.Drawing.mleader_styles.new()

class ezdxf.entities.**MLeaderStyle**

dxf.**align_space**
: unknown meaning

dxf.**arrow_head_handle**
: handle of default arrow head, see also *ezdxf.render.arrows* module, by default no handle is set, which mean default arrow "closed filled"

dxf.**arrow_head_size**
: default arrow head size in drawing units, default is 4.0

dxf.**block_color**
: default block color as ;term:*raw color* value, default is BY_BLOCK_RAW_VALUE

dxf.**block_connection_type**

0	center extents
1	insertion point

dxf.**block_record_handle**
handle to block record of the BLOCK content, not set by default

dxf.**block_rotation**
default BLOCK rotation in radians, default is 0.0

dxf.**block_scale_x**
default block x-axis scale factor, default is 1.0

dxf.**block_scale_y**
default block y-axis scale factor, default is 1.0

dxf.**block_scale_z**
default block z-axis scale factor, default is 1.0

dxf.**break_gap_size**
default break gap size, default is 3.75

dxf.**char_height**
default MTEXT char height, default is 4.0

dxf.**content_type**

0	none
1	BLOCK
2	MTEXT
3	TOLERANCE

default is MTEXT (2)

dxf.**default_text_content**
default MTEXT content as string, default is “”

dxf.**dogleg_length**
default dogleg length, default is 8.0

dxf.**draw_leader_order_type**
unknown meaning

dxf.**draw_mleader_order_type**
unknown meaning

dxf.**first_segment_angle_constraint**
angle of fist leader segment in radians, default is 0.0

dxf.**has_block_rotation**

dxf.**has_block_scaling**

dxf.**has_dogleg**
default is 1

dxf.**has_landing**
default is 1

dxf.**is_annotative**
default is 0

dxf.**landing_gap**
default landing gap size, default is 2.0

dxf.**leader_line_color**
default leader line color as *raw-color* value, default is BY_BLOCK_RAW_VALUE

dxf.**leader_linetype_handle**
handle of default leader linetype

dxf.**leader_lineweight**
default leader lineweight, default is LINEWEIGHT_BYBLOCK

dxf.**leader_type**

0	invisible
1	straight line leader
2	spline leader

default is 1

dxf.**max_leader_segments_points**
max count of leader segments, default is 2

dxf.**name**
MLEADERSTYLE name

dxf.**overwrite_property_value**
unknown meaning

dxf.**scale**
overall scaling factor, default is 1.0

dxf.**second_segment_angle_constraint**
angle of fist leader segment in radians, default is 0.0

dxf.**text_align_always_left**
use always left side to attach leaders, default is 0

dxf.**text_alignment_type**
unknown meaning - its not the MTEXT attachment point!

dxf.**text_angle_type**

0	text angle is equal to last leader line segment angle
1	text is horizontal
2	text angle is equal to last leader line segment angle, but potentially rotated by 180 degrees so the right side is up for readability.

default is 1

dxf.**text_attachment_direction**

defines whether the leaders attach to the left & right of the content BLOCK/MTEXT or attach to the top & bottom:

0	horizontal - left & right of content
1	vertical - top & bottom of content

default is 0

dxf.**text_bottom_attachment_type**

9	center
10	overline and center

default is 9

dxf.**text_color**

default MTEXT color as *raw-color* value, default is BY_BLOCK_RAW_VALUE

dxf.**text_left_attachment_type**

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

dxf.**text_right_attachment_type**

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

dxf.**text_style_handle**

handle of the default MTEXT text style, not set by default, which means “Standard”

dxf.**text_top_attachment_type**

9	center
10	overline and center

Placeholder

The ACDBPLACEHOLDER object for internal usage.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'ACDBPLACEHOLDER'
Factory function	*ezdxf.sections.objects.ObjectsSection.add_placeholder()*

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.**Placeholder**

PlotSettings

All PLOTSETTINGS attributes are part of the *DXFLayout* entity, I don't know if this entity also appears as standalone entity.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'PLOTSETTINGS'
Factory function	internal data structure

class ezdxf.entities.**PlotSettings**

dxf.**page_setup_name**
default is ""

dxf.**plot_configuration_file**
default is "Adobe PDF"

dxf.**paper_size**
default is "A3"

dxf.**plot_view_name**
default is ""

dxf.**left_margin**
default is 7.5 mm

dxf.**bottom_margin**
default is 20 mm

dxf.**right_margin**
default is 7.5 mm

dxf.**top_margin**
default is 20 mm

dxf.**paper_width**
default is 420 mm

dxf.**paper_height**
default is 297 mm

dxf.**plot_origin_x_offset**
default is 0

dxf.**plot_origin_y_offset**
default is 0

dxf.**plot_window_x1**
default is 0

dxf.**plot_window_y1**
default is 0

dxf.**plot_window_x2**
default is 0

dxf.**plot_window_y2**
default is 0

dxf.**scale_numerator**
default is 1

dxf.**scale_denominator**
default is 1

dxf.**plot_layout_flags**

1	plot viewport borders
2	show plot-styles
4	plot centered
8	plot hidden == hide paperspace entities?
16	use standard scale
32	plot with plot-styles
64	scale lineweights
128	plot entity lineweights
512	draw viewports first
1024	model type
2048	update paper
4096	zoom to paper on update
8192	initializing
16384	prev plot-init

default is 688

dxf.**plot_paper_units**

0	Plot in inches
1	Plot in millimeters
2	Plot in pixels

dxf.**plot_rotation**

0	No rotation
1	90 degrees counterclockwise
2	Upside-down
3	90 degrees clockwise

dxf.**plot_type**

0	Last screen display
1	Drawing extents
2	Drawing limits
3	View specified by code 6
4	Window specified by codes 48, 49, 140, and 141
5	Layout information

dxf.**current_style_sheet**

default is “”

dxf.**standard_scale_type**

0	Scaled to Fit
1	1/128”=1’
2	1/64”=1’
3	1/32”=1’
4	1/16”=1’
5	3/32”=1’
6	1/8”=1’
7	3/16”=1’
8	1/4”=1’
9	3/8”=1’
10	1/2”=1’
11	3/4”=1’
12	1”=1’
13	3”=1’
14	6”=1’
15	1’=1’
16	1:1
17	1:2
18	1:4
19	1:8
20	1:10
21	1:16
22	1:20

continues on next page

Table 1 – continued from previous page

23	1:30
24	1:40
25	1:50
26	1:100
27	2:1
28	4:1
29	8:1
30	10:1
31	100:1
32	1000:1

dxf.**shade_plot_mode**

0	As Displayed
1	Wireframe
2	Hidden
3	Rendered

dxf.**shade_plot_resolution_level**

0	Draft
1	Preview
2	Normal
3	Presentation
4	Maximum
5	Custom

dxf.**shade_plot_custom_dpi**
default is 300

dxf.**unit_factor**
default is 1

dxf.**paper_image_origin_x**
default is 0

dxf.**paper_image_origin_y**
default is 0

dxf.**shade_plot_handle**

Sun

The SUN entity defines properties of the sun.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	'SUN'
Factory function	creating a new SUN entity is not supported

class ezdxf.entities.**Sun**

dxf.**version**
Current version is 1.

dxf.**status**
on = 1 or off = 0

dxf.**color**
AutoCAD Color Index (ACI) value of the sun.

dxf.**true_color**
true-color value of the sun.

dxf.**intensity**
Intensity value in the range of [0, 1]. (float)

dxf.**julian_day**
use *calendardate()* to convert dxf.julian_day to datetime.datetime object.

dxf.**time**
Day time in seconds past midnight. (int)

dxf.**daylight_savings_time**

dxf.**shadows**

0	Sun do not cast shadows
1	Sun do cast shadows

dxf.**shadow_type**

dxf.**shadow_map_size**

dxf.**shadow_softness**

UnderlayDefinition

UnderlayDefinition (DXF Reference) defines an underlay file, which can be placed by the *Underlay* entity.

Subclass of	*ezdxf.entities.DXFObject*
DXF type	internal base class
Factory function (1)	*ezdxf.document.Drawing.add_underlay_def()*
Factory function (2)	*ezdxf.sections.objects.ObjectsSection.add_underlay_def()*

class ezdxf.entities.**UnderlayDefinition**

Base class of *PdfDefinition*, *DwfDefinition* and *DgnDefinition*

dxf.**filename**

Relative (to the DXF file) or absolute path to the underlay file as string.

dxf.**name**

Defines which part of the underlay file to display.

"pdf"	PDF page number
"dgn"	always "default"
"dwf"	?

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

PdfDefinition

Subclass of	*ezdxf.entities.UnderlayDefinition*
DXF type	'PDFDEFINITION'
Factory function (1)	*ezdxf.document.Drawing.add_underlay_def()*
Factory function (2)	*ezdxf.sections.objects.ObjectsSection.add_underlay_def()*

class ezdxf.entities.**PdfDefinition**

PDF underlay file.

DwfDefinition

Subclass of	*ezdxf.entities.UnderlayDefinition*
DXF type	'DWFDEFINITION'
Factory function (1)	*ezdxf.document.Drawing.add_underlay_def()*
Factory function (2)	*ezdxf.sections.objects.ObjectsSection.add_underlay_def()*

class ezdxf.entities.**DwfDefinition**

DWF underlay file.

DgnDefinition

Subclass of	*ezdxf.entities.UnderlayDefinition*
DXF type	'DGNDEFINITION'
Factory function (1)	*ezdxf.document.Drawing.add_underlay_def()*
Factory function (2)	*ezdxf.sections.objects.ObjectsSection.add_underlay_def()*

class ezdxf.entities.**DgnDefinition**

DGN underlay file.

XRecord

Important class for storing application defined data in DXF files.

The XRECORD entities are used to store and manage arbitrary data. They are composed of DXF group codes ranging from 1 through 369. This object is similar in concept to XDATA but is not limited by size or order.

To reference a XRECORD by an DXF entity, store the handle of the XRECORD in the XDATA section, application defined data or the `ExtensionDict` of the DXF entity.

See also:

- *Extended Data (XDATA)*
- *Extension Dictionary*
- *Storing Custom Data in DXF Files*

Subclass of	*ezdxf.entities.DXFObject*
DXF type	`'XRECORD'`
Factory function	*ezdxf.sections.objects.ObjectsSection.add_xrecord()*

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.**XRecord**

dxf.**cloning**

Duplicate record cloning flag (determines how to merge duplicate entries, ignored by *ezdxf*):

0	not applicable
1	keep existing
2	use clone
3	<xref>0<name>
4	0<name>
5	Unmangle name

tags

Raw DXF tag container *Tags*. Be careful *ezdxf* does not validate the content of XRECORDS.

clear() → None

Remove all DXF tags.

reset(*tags: Iterable[*DXFTag | *tuple[int, Any]]*) → None

Reset DXF tags.

extend(*tags: Iterable[*DXFTag | *tuple[int, Any]]*) → None

Extend DXF tags.

Extended Data (XDATA)

Extended data (XDATA) is a DXF tags structure to store arbitrary data in DXF entities. The XDATA is associated to an *AppID* and only one tag list is supported for each AppID per entity.

Warning: Low level usage of XDATA is an advanced feature, it is the responsibility of the programmer to create valid XDATA structures. Any errors can invalidate the DXF file!

This section shows how to store DXF tags directly in DXF entity but there is also a more user friendly and safer way to store custom XDATA in DXF entities:

- *XDataUserList*
- *XDataUserDict*

Use the high level methods of *DXFEntity* to manage XDATA tags.

- *has_xdata()*
- *get_xdata()*
- *set_xdata()*

Get XDATA tags as a *ezdxf.lldxf.tags.Tags* data structure, **without** the mandatory first tag (1001, AppID):

```
if entity.has_xdata("EZDXF"):
    tags = entity.get_xdata("EZDXF")

# or use alternatively:
try:
    tags = entity.get_xdata("EZDXF")
except DXFValueError:
    # XDATA for "EZDXF" does not exist
    ...
```

Set DXF tags as list of (group code, value) tuples or as *ezdxf.lldxf.tags.Tags* data structure, valid DXF tags for XDATA are documented in the section about the *Extended Data* internals. The mandatory first tag (1001, AppID) is inserted automatically if not present.

Set only new XDATA tags:

```
if not entity.has_xdata("EZDXF"):
    entity.set_xdata("EZDXF", [(1000, "MyString")])
```

Replace or set new XDATA tags:

```
entity.set_xdata("EZDXF", [(1000, "MyString")])
```

See also:

- Tutorial: *Storing Custom Data in DXF Files*
- Internals about *Extended Data* tags
- Internal XDATA management class: *XData*
- DXF R2018 Reference

Application-Defined Data (AppData)

The application-defined data feature is not very well documented in the DXF reference, so usage as custom data store is not recommended. AutoCAD uses these feature to store the handle to the extension dictionary (*ExtensionDict*) of a DXF entity and the handles to the persistent reactors (*Reactors*) of a DXF entity.

Use the high level methods of *DXFEntity* to manage application-defined data tags.

- *has_app_data()*
- *get_app_data()*
- *set_app_data()*
- *discard_app_data()*

Hint: Ezdxf uses special classes to manage the extension dictionary and the reactor handles. These features cannot be accessed by the methods above.

Set application-defined data:

```
entity.set_app_data("YOURAPPID", [(1, "DataString")]))
```

Setting the content tags can contain the opening structure tag (102, "{YOURAPPID") and the closing tag (102, "}"), but doesn't have to. The returned *Tags* objects does not contain these structure tags. Which tags are valid for application-defined data is not documented.

The AppID has to have an entry in the AppID table.

Get application-defined data:

```
if entity.has_app_data("YOURAPPID"):
    tags = entity.get_app_data("YOURAPPID")

# tags content is [DXFTag(1, 'DataString')]
```

See also:

- Internals about *Application-Defined Codes*
- Internal AppData management class: *AppData*

Extension Dictionary

Every entity can have an extension dictionary, which can reference arbitrary DXF objects from the OBJECTS section but not graphical entities. Using this mechanism, several applications can attach data to the same entity. The usage of extension dictionaries is more complex than *Extended Data (XDATA)* but also more flexible with higher capacity for adding data.

Use the high level methods of *DXFEntity* to manage extension dictionaries.

- *has_extension_dict()*
- *get_extension_dict()*
- *new_extension_dict()*
- *discard_extension_dict()*

The main data storage objects referenced by extension dictionaries are:

- *Dictionary*, structural container
- *DictionaryVar*, stores a single string
- *XRecord*, stores arbitrary data

See also:

- Tutorial: *Storing Custom Data in DXF Files*

class ezdxf.entities.xdict.**ExtensionDict**

Internal management class for extension dictionaries.

See also:

- Underlying DXF *Dictionary* class
- DXF Internals: *Extension Dictionary*
- DXF R2018 Reference

property is_alive

Returns `True` if the underlying *Dictionary* object is not deleted.

__contains__(*key: str*)

Return *key* in self.

__getitem__(*key: str*)

Get self[key].

__setitem__(*key: str*, *value*)

Set self[key] to value.

Only DXF objects stored in the OBJECTS section are allowed as content of the extension dictionary. DXF entities stored in layouts are not allowed.

Raises
DXFTypeError – invalid DXF type

__delitem__(*key: str*)

Delete self[key], destroys referenced entity.

__len__()

Returns count of extension dictionary entries.

get(*key: str*, *default=None*) → *DXFEntity* | None

Return extension dictionary entry *key*.

keys()

Returns a `KeysView` of all extension dictionary keys.

items()

Returns an `ItemsView` for all extension dictionary entries as (key, entity) pairs. An entity can be a handle string if the entity does not exist.

discard(*key: str*) → None

Discard extension dictionary entry *key*.

add_dictionary(*name: str*, *hard_owned: bool = True*) → *Dictionary*

Create a new *Dictionary* object as extension dictionary entry *name*.

add_dictionary_var(*name: str*, *value: str*) → *DictionaryVar*

Create a new `DictionaryVar` object as extension dictionary entry *name*.

add_xrecord(*name: str*) → *XRecord*

Create a new `XRecord` object as extension dictionary entry *name*.

link_dxf_object(*name: str*, *obj:* DXFObject) → None

Link *obj* to the extension dictionary as entry *name*.

Linked objects are owned by the extensions dictionary and therefore cannot be a graphical entity, which have to be owned by a `BaseLayout`.

Raises

DXFTypeError – *obj* has invalid DXF type

destroy()

Destroy the underlying `Dictionary` object.

Reactors

Persistent reactors are optional object handles of objects registering themselves as reactors on an object. Any DXF object or DXF entity may have reactors.

Use the high level methods of `DXFEntity` to manage persistent reactor handles.

- `has_reactors()`
- `get_reactors()`
- `set_reactors()`
- `append_reactor_handle()`
- `discard_reactor_handle()`

Ezdxf keeps these reactors only up to date, if this is absolute necessary according to the DXF reference.

See also:

- Internals about *Persistent Reactors*
- Internal Reactors management class: `Reactors`

Block Reference Management

The package *ezdxf* is not designed as a CAD library and does not automatically monitor all internal changes. This enables faster entity processing at the cost of an unknown state of the DXF document.

In order to carry out precise BLOCK reference management, i.e. to handle dependencies or to delete unused BLOCK definition, the block reference status (counter) must be acquired explicitly by the package user. All block reference management structures must be explicitly recreated each time the document content is changed. This is not very efficient, but it is safe.

Warning: And even with all this careful approach, it is always possible to destroy a DXF document by deleting an absolutely necessary block definition.

Always remember that *ezdxf* is not intended or suitable as a basis for a CAD application!

class ezdxf.blkrefs.**BlockDefinitionIndex**(*doc:* Drawing)

Index of all *BlockRecord* entities representing real BLOCK definitions, excluding all *BlockRecord* entities defining model space or paper space layouts. External references (XREF) and XREF overlays are included.

property block_records: Iterator[*BlockRecord*]

Returns an iterator of all *BlockRecord* entities representing BLOCK definitions.

rebuild()

Rebuild index from scratch.

has_handle(*handle: str*) → bool

Returns `True` if a *BlockRecord* for the given block record handle exist.

by_handle(*handle: str*) → *BlockRecord* | None

Returns the *BlockRecord* for the given block record handle or `None`.

has_name(*name: str*) → bool

Returns `True` if a *BlockRecord* for the given block name exist.

by_name(*name: str*) → *BlockRecord* | None

Returns *BlockRecord* for the given block name or `None`.

class ezdxf.blkrefs.**BlockReferenceCounter**(*doc:* Drawing, *index:* BlockDefinitionIndex | *None* = *None*)

Counts all block references in a DXF document.

Check if a block is referenced by any entity or any resource (DIMSYTLE, MLEADERSTYLE) in a DXF document:

```
import ezdxf
from ezdxf.blkrefs import BlockReferenceCounter

doc = ezdxf.readfile("your.dxf")
counter = BlockReferenceCounter(doc)
count = counter.by_name("XYZ")
print(f"Block 'XYZ' if referenced {count} times.")
```

by_handle(*handle: str*) → int

Returns the block reference count for a given *BlockRecord* handle.

by_name(*block_name: str*) → int

Returns the block reference count for a given block name.

Const

The module *ezdxf.lldxf.const*, is also accessible from the `ezdxf` namespace:

```
from ezdxf.lldxf.const import DXF12
import ezdxf

print(DXF12)
print(ezdxf.const.DXF12)
```

DXF Version Strings

Name	Version	Alias
DXF9	“AC1004”	“R9”
DXF10	“AC1006”	“R10”
DXF12	“AC1009”	“R12”
DXF13	“AC1012”	“R13”
DXF14	“AC1014”	“R14”
DXF2000	“AC1015”	“R2000”
DXF2004	“AC1018”	“R2004”
DXF2007	“AC1021”	“R2007”
DXF2010	“AC1024”	“R2010”
DXF2013	“AC1027”	“R2013”
DXF2018	“AC1032”	“R2018”

Exceptions

class ezdxf.lldxf.const.**DXFError**

Base exception for all *ezdxf* exceptions.

class ezdxf.lldxf.const.**DXFStructureError**(*DXFError*)

class ezdxf.lldxf.const.**DXFVersionError**(*DXFError*)

Errors related to features not supported by the chosen DXF Version

class ezdxf.lldxf.const.**DXFValueError**(*DXFError*)

class ezdxf.lldxf.const.**DXFInvalidLineType**(*DXFValueError*)

class ezdxf.lldxf.const.**DXFBlockInUseError**(*DXFValueError*)

class ezdxf.lldxf.const.**DXFKeyError**(*DXFError*)

class ezdxf.lldxf.const.**DXFUndefinedBlockError**(*DXFKeyError*)

class ezdxf.lldxf.const.**DXFAttributeError**(*DXFError*)

class ezdxf.lldxf.const.**DXFIndexError**(*DXFError*)

class ezdxf.lldxf.const.**DXFTypeError**(*DXFError*)

class ezdxf.lldxf.const.**DXFTableEntryError**(*DXFValueError*)

6.9.3 DXF Entity Creation

Layout Factory Methods

Recommended way to create DXF entities.

For all supported entities exist at least one factory method in the *ezdxf.layouts.BaseLayout* class. All factory methods have the prefix: `add_...`

```
import ezdxf

doc = ezdxf.new()
msp = doc.modelspace()
msp.add_line((0, 0, 0), (3, 0, 0), dxfattribs={"color": 2})
```

Thematic Index of Layout Factory Methods

DXF Primitives

- *add_3dface()*
- *add_arc()*
- *add_circle()*
- *add_ellipse()*
- *add_hatch()*
- *add_helix()*
- *add_image()*
- *add_leader()*
- *add_line()*
- *add_lwpolyline()*
- *add_mesh()*
- *add_mline()*
- *add_mpolygon()*
- *add_multileader_mtext()*
- *add_multileader_block()*
- *add_point()*
- *add_polyface()*
- *add_polyline2d()*
- *add_polyline3d()*
- *add_polymesh()*
- *add_ray()*
- *add_shape()*
- *add_solid()*
- *add_trace()*
- *add_wipeout()*
- *add_xline()*

Text Entities

- *add_attdef()*
- *add_mtext_dynamic_auto_height_columns()*
- *add_mtext_dynamic_manual_height_columns()*
- *add_mtext_static_columns()*
- *add_mtext()*
- *add_text()*

Spline Entity

- *add_cad_spline_control_frame()*
- *add_open_spline()*
- *add_rational_spline()*
- *add_spline_control_frame()*
- *add_spline()*

Block References and Underlays

- add_arrow_blockref()
- *add_auto_blockref()*
- *add_blockref()*
- *add_underlay()*

Viewport Entity

Only available in paper space layouts.

- add_viewport()

Dimension Entities

Linear Dimension

- *add_aligned_dim()*
- *add_linear_dim()*
- *add_multi_point_linear_dim()*

Radius and Diameter Dimension

- *add_diameter_dim_2p()*
- *add_diameter_dim()*
- *add_radius_dim_2p()*

- *add_radius_dim_cra()*
- *add_radius_dim()*

Angular Dimension

- *add_angular_dim_2l()*
- *add_angular_dim_3p()*
- *add_angular_dim_arc()*
- *add_angular_dim_cra()*

Arc Dimension

- *add_arc_dim_3p()*
- *add_arc_dim_arc()*
- *add_arc_dim_cra()*

Ordinate Dimension

- *add_ordinate_dim()*
- *add_ordinate_x_dim()*
- *add_ordinate_y_dim()*

Miscellaneous

- *add_entity()*
- *add_foreign_entity()*
- add_arrow()

ACIS Entities

The creation of the required *ACIS* data has do be done by an external library!

- *add_3dsolid()*
- *add_body()*
- *add_extruded_surface()*
- *add_lofted_surface()*
- *add_region()*
- *add_revolved_surface()*
- *add_surface()*
- *add_swept_surface()*

See also:

Layout base class: *BaseLayout*

Factory Functions

Alternative way to create DXF entities for advanced *ezdxf* users.

The `ezdxf.entities.factory` module provides the `new()` function to create new DXF entities by their DXF name and a dictionary of DXF attributes. This will bypass the validity checks in the factory methods of the *BaseLayout* class.

This new created entities are virtual entities which are not assigned to any DXF document nor to any layout. Add the entity to a layout (and document) by the layout method *add_entity()*.

```
import ezdxf
from ezdxf.entities import factory

doc = ezdxf.new()
msp = doc.modelspace()
line = factory.new(
    "LINE",
    dxfattribs={
        "start": (0, 0, 0),
        "end": (3, 0, 0),
        "color": 2,
    },
)
msp.add_entity(line)
```

Direct Object Instantiation

For advanced developers with knowledge about the internal design of *ezdxf*.

Import the entity classes from sub-package *ezdxf.entities* and instantiate them. This will bypass the validity checks in the factory methods of the *BaseLayout* class and maybe additional required setup procedures for some entities - **study the source code!**.

Warning: A refactoring of the internal *ezdxf* structures will break your code.

This new created entities are virtual entities which are not assigned to any DXF document nor to any layout. Add the entity to a layout (and document) by the layout method *add_entity()*.

```
import ezdxf
from ezdxf.entities import Line

doc = ezdxf.new()
msp = doc.modelspace()
line = Line.new(
    dxfattribs={
        "start": (0, 0, 0),
        "end": (3, 0, 0),
        "color": 2,
    }
)
msp.add_entity(line)
```

6.9.4 Enums

TextEntityAlignment

class ezdxf.enums.**TextEntityAlignment**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Text alignment enum for the *Text*, *Attrib* and *AttDef* entities.

LEFT

CENTER

RIGHT

ALIGNED

MIDDLE

FIT

BOTTOM_LEFT

BOTTOM_CENTER

BOTTOM_RIGHT

MIDDLE_LEFT

MIDDLE_CENTER

MIDDLE_RIGHT

TOP_LEFT

TOP_CENTER

TOP_RIGHT

MTextEntityAlignment

class ezdxf.enums.**MTextEntityAlignment**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Text alignment enum for the *MText* entity.

TOP_LEFT

TOP_CENTER

TOP_RIGHT

MIDDLE_LEFT

MIDDLE_CENTER

MIDDLE_RIGHT

BOTTOM_LEFT

BOTTOM_CENTER

BOTTOM_RIGHT

MTextParagraphAlignment

class ezdxf.enums.**MTextParagraphAlignment**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

DEFAULT

LEFT

RIGHT

CENTER

JUSTIFIED

DISTRIBUTED

MTextFlowDirection

class ezdxf.enums.**MTextFlowDirection**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

LEFT_TO_RIGHT

TOP_TO_BOTTOM

BY_STYLE

MTextLineAlignment

class ezdxf.enums.**MTextLineAlignment**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

BOTTOM

MIDDLE

TOP

MTextStroke

class ezdxf.enums.**MTextStroke**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Combination of flags is supported: UNDERLINE + STRIKE_THROUGH

UNDERLINE

STRIKE_THROUGH

OVERLINE

MTextLineSpacing

class ezdxf.enums.**MTextLineSpacing**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

AT_LEAST

EXACT

MTextBackgroundColor

class ezdxf.enums.**MTextBackgroundColor**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

OFF

COLOR

WINDOW

CANVAS

InsertUnits

class ezdxf.enums.**InsertUnits**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Unitless

Inches

Feet

Miles

Millimeters

Centimeters

Meters

Kilometers

Microinches

Mils

Yards

Angstroms

Nanometers

Microns

Decimeters

Decameters

Hectometers

Gigameters

AstronomicalUnits

Lightyears

Parsecs

USSurveyFeet

USSurveyInch

USSurveyYard

USSurveyMile

Measurement

class ezdxf.enums.**Measurement**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Imperial

Metric

LengthUnits

class ezdxf.enums.**LengthUnits**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Scientific

Decimal

Engineering

Architectural

Fractional

AngularUnits

class ezdxf.enums.**AngularUnits**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

DecimalDegrees

DegreesMinutesSeconds

Grad

Radians

SortEntities

class ezdxf.enums.**SortEntities**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

DISABLE

SELECTION

Sorts for object selection

SNAP

Sorts for object snap

REDRAW

Sorts for redraws; obsolete

MSLIDE

Sorts for MSLIDE command slide creation; obsolete

REGEN

Sorts for REGEN commands

PLOT

Sorts for plotting

POSTSCRIPT

Sorts for PostScript output; obsolete

ACI

class ezdxf.enums.**ACI**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

AutoCAD Color Index

BYBLOCK

BYLAYER

BYOBJECT

RED

YELLOW

GREEN

CYAN

BLUE

MAGENTA

BLACK

WHITE

GRAY

LIGHT_GRAY

EndCaps

class ezdxf.enums.**EndCaps**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Lineweight end caps setting for new objects.

NONE

ROUND

ANGLE

SQUARE

JoinStyle

class ezdxf.enums.**JoinStyle**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Lineweight joint setting for new objects.

NONE

ROUND

ANGLE

FLAT

6.9.5 Colors

Colors Module

This module provides functions and constants to manage all kinds of colors in DXF documents.

Converter Functions

ezdxf.colors.**rgb2int**(*rgb: Tuple[int, int, int]*) → int

Combined integer value from (r, g, b) tuple.

ezdxf.colors.**int2rgb**(*value: int*) → Tuple[int, int, int]

Split RGB integer *value* into (r, g, b) tuple.

ezdxf.colors.**aci2rgb**(*index: int*) → Tuple[int, int, int]

Convert *AutoCAD Color Index (ACI)* into (r, g, b) tuple, based on default AutoCAD colors.

ezdxf.colors.**luminance**(*color: Tuple[int, int, int]*) → float

Returns perceived luminance for an RGB color in the range [0.0, 1.0] from dark to light.

ezdxf.colors.**decode_raw_color**(*value: int*) → tuple[int, Union[int, Tuple[int, int, int]]]

Decode *raw-color* value as tuple(type, Union[aci, (r, g, b)]), the true color value is a (r, g, b) tuple.

ezdxf.colors.**decode_raw_color_int**(*value: int*) → tuple[int, int]

Decode *raw-color* value as tuple(type, int), the true color value is a 24-bit int value.

ezdxf.colors.**encode_raw_color**(*value: int | Tuple[int, int, int]*) → int

Encode *true-color* value or *AutoCAD Color Index (ACI)* color value into a :term: *raw color* value.

ezdxf.colors.**transparency2float**(*value: int*) → float

Returns transparency value as float from 0 to 1, 0 for no transparency (opaque) and 1 for 100% transparency.

Parameters

value – DXF integer transparency value, 0 for 100% transparency and 255 for opaque

ezdxf.colors.**float2transparency**(*value: float*) → int

Returns DXF transparency value as integer in the range from 0 to 255, where 0 is 100% transparent and 255 is opaque.

Parameters

value – transparency value as float in the range from 0 to 1, where 0 is opaque and 1 is 100% transparent.

ACI Color Values

Common *AutoCAD Color Index (ACI)* values, also accessible as IntEnum `ezdxf.enums.ACI`

BYBLOCK	0
BYLAYER	256
BYOBJECT	257
RED	1
YELLOW	2
GREEN	3
CYAN	4
BLUE	5
MAGENTA	6
BLACK (on light background)	7
WHITE (on dark background)	7
GRAY	8
LIGHT_GRAY	9

Default Palettes

Default color mappings from *AutoCAD Color Index (ACI)* to *true-color* values.

model space	DXF_DEFAULT_COLORS
paper space	DXF_DEFAULT_PAPERSPACE_COLORS

Raw Color Types

COLOR_TYPE_BY_LAYER	0xC0
COLOR_TYPE_BY_BLOCK	0xC1
COLOR_TYPE_RGB	0xC2
COLOR_TYPE_ACI	0xC3
COLOR_TYPE_WINDOW_BG	0xC8

Raw Color Vales

BY_LAYER_RAW_VALUE	-1073741824
BY_BLOCK_RAW_VALUE	-1056964608
WINDOW_BG_RAW_VALUE	-939524096

Transparency Values

OPAQUE	0x20000FF
TRANSPARENCY_10	0x20000E5
TRANSPARENCY_20	0x20000CC
TRANSPARENCY_30	0x20000B2
TRANSPARENCY_40	0x2000099
TRANSPARENCY_50	0x200007F
TRANSPARENCY_60	0x2000066
TRANSPARENCY_70	0x200004C
TRANSPARENCY_80	0x2000032
TRANSPARENCY_90	0x2000019
TRANSPARENCY_BYBLOCK	0x1000000

6.9.6 Data Query

See also:

For usage of the query features see the tutorial: *Tutorial for Getting Data from DXF Files*

Entity Query String

```
QueryString := EntityQuery ("[" AttribQuery "]" "i"?)*
```

The query string is the combination of two queries, first the required entity query and second the optional attribute query, enclosed in square brackets, append `'i'` after the closing square bracket to ignore case for strings.

Entity Query

The entity query is a whitespace separated list of DXF entity names or the special name `'*'`. Where `'*'` means all DXF entities, exclude some entity types by appending their names with a preceding `!` (e.g. all entities except LINE = `'* !LINE'`). All DXF names have to be uppercase.

Attribute Query

The *optional* attribute query is a boolean expression, supported operators are:

- not (!): !term is true, if term is false
- and (&): term & term is true, if both terms are true
- or (|): term | term is true, if one term is true
- and arbitrary nested round brackets
- append (i) after the closing square bracket to ignore case for strings

Attribute selection is a term: "name comparator value", where name is a DXF entity attribute in lowercase, value is a integer, float or double quoted string, valid comparators are:

- == equal "value"
- != not equal "value"
- < lower than "value"
- <= lower or equal than "value"
- > greater than "value"
- >= greater or equal than "value"
- ? match regular expression "value"
- !? does not match regular expression "value"

Query Result

The *EntityQuery* class is the return type of all `query()` methods. *EntityQuery* contains all DXF entities of the source collection, which matches one name of the entity query AND the whole attribute query. If a DXF entity does not have or support a required attribute, the corresponding attribute search term is `False`.

examples:

- `LINE[text ? ".*"]`: always empty, because the LINE entity has no text attribute.
- `LINE CIRCLE[layer=="construction"]`: all LINE and CIRCLE entities with layer == `"construction"`
- `*[!(layer=="construction" & color<7)]`: all entities except those with layer == `"construction"` and color < `7`
- `*[layer=="construction"]i`, (ignore case) all entities with layer == `"construction"` | `"Construction"` | `"ConStruction"` ...

EntityQuery Class

class ezdxf.query.**EntityQuery**

The *EntityQuery* class is a result container, which is filled with DXF entities matching the query string. It is possible to add entities to the container (extend), remove entities from the container and to filter the container. Supports the standard Python Sequence methods and protocols. Does not remove automatically destroyed entities (entities deleted by calling method `destroy()`), the method *purge()* has to be called explicitly to remove the destroyed entities.

first

First entity or `None`.

last

Last entity or `None`.

__len__() → int

Returns count of DXF entities.

__getitem__(*item*)

Returns DXFEntity at index *item*, supports negative indices and slicing. Returns all entities which support a specific DXF attribute, if *item* is a DXF attribute name as string.

__setitem__(*key*, *value*)

Set the DXF attribute *key* for all supported DXF entities to *value*.

__delitem__(*key*)

Discard the DXF attribute *key* from all supported DXF entities.

__eq__(*other*)

Equal selector (self == other). Returns all entities where the selected DXF attribute is equal to *other*.

__ne__(*other*)

Not equal selector (self != other). Returns all entities where the selected DXF attribute is not equal to *other*.

__lt__(*other*)

Less than selector (self < other). Returns all entities where the selected DXF attribute is less than *other*.

Raises

TypeError – for vector based attributes like *center* or *insert*

__le__(*other*)

Less equal selector (self <= other). Returns all entities where the selected DXF attribute is less or equal *other*.

Raises

TypeError – for vector based attributes like *center* or *insert*

__gt__(*other*)

Greater than selector (self > other). Returns all entities where the selected DXF attribute is greater than *other*.

Raises

TypeError – for vector based attributes like *center* or *insert*

__ge__(*other*)

Greater equal selector (self >= other). Returns all entities where the selected DXF attribute is greater or equal *other*.

Raises

TypeError – for vector based attributes like *center* or *insert*

match(*pattern: str*) → *EntityQuery*

Returns all entities where the selected DXF attribute matches the regular expression *pattern*.

Raises

TypeError – for non-string based attributes

__or__(*other*)

Union operator, see *union()*.

__and__(*other*)

Intersection operator, see *intersection()*.

__sub__(*other*)

Difference operator, see *difference()*.

__xor__(*other*)

Symmetric difference operator, see *symmetric_difference()*.

__iter__() → Iterator[DXFEntity]

Returns iterable of DXFEntity objects.

purge() → *EntityQuery*

Remove destroyed entities.

extend(*entities: Iterable[DXFEntity]*, *query: str = '*'*) → *EntityQuery*

Extent the *EntityQuery* container by entities matching an additional query.

remove(*query: str = '*'*) → *EntityQuery*

Remove all entities from *EntityQuery* container matching this additional query.

query(*query: str = '*'*) → *EntityQuery*

Returns a new *EntityQuery* container with all entities matching this additional query.

Raises

pyparsing.ParseException – query string parsing error

groupby(*dxfattrib: str = ''*, *key: Callable[[DXFEntity], Hashable] | None = None*) → dict[Hashable, list[ezdxf.entities.dxfentity.DXFEntity]]

Returns a dict of entity lists, where entities are grouped by a DXF attribute or a key function.

Parameters

- **dxfattrib** – grouping DXF attribute as string like `'layer'`
- **key** – key function, which accepts a DXFEntity as argument, returns grouping key of this entity or `None` for ignore this object. Reason for ignoring: a queried DXF attribute is not supported by this entity

filter(*func: Callable[[DXFEntity], bool]*) → *EntityQuery*

Returns a new *EntityQuery* with all entities from this container for which the callable *func* returns `True`.

Build your own operator to filter by attributes which are not DXF attributes or to build complex queries:

```
result = msp.query().filter(
    lambda e: hasattr(e, "rgb") and e.rbg == (0, 0, 0)
)
```

union(*other:* EntityQuery) → *EntityQuery*

Returns a new *EntityQuery* with entities from *self* and *other*. All entities are unique - no duplicates.

intersection(*other:* EntityQuery) → *EntityQuery*

Returns a new *EntityQuery* with entities common to *self* and *other*.

difference(*other:* EntityQuery) → *EntityQuery*

Returns a new *EntityQuery* with all entities from *self* that are not in *other*.

symmetric_difference(*other:* EntityQuery) → *EntityQuery*

Returns a new *EntityQuery* with entities in either *self* or *other* but not both.

Extended EntityQuery Features

The `[]` operator got extended features in version 0.18, until then the *EntityQuery* implemented the `__getitem__()` interface like a sequence to get entities from the container:

```
result = msp.query(...)
first = result[0]
last = result[-1]
sequence = result[1:-2]  # returns not an EntityQuery container!
```

Now the `__getitem__()` function accepts also a DXF attribute name and returns all entities which support this attribute, this is the base for supporting queries by relational operators. More on that later.

The `__setitem__()` method assigns a DXF attribute to all supported entities in the *EntityQuery* container:

```
result = msp.query(...)
result["layer"] = "MyLayer"
```

Entities which do not support an attribute are silently ignored:

```
result = msp.query(...)
result["center"] = (0, 0)  # set center only of CIRCLE and ARC entities
```

The `__delitem__()` method discards DXF attributes from all entities in the *EntityQuery* container:

```
result = msp.query(...)
# reset the layer attribute from all entities in container result to the
# default layer "0"
del result["layer"]
```

Descriptors for DXF Attributes

For some basic DXF attributes exist descriptors in the *EntityQuery* class:

- `layer`: layer name as string
- `color`: *AutoCAD Color Index (ACI)*, see *ezdxf.colors*
- `linetype`: linetype as string
- `ltscale`: linetype scaling factor as float value
- `lineweight`: *Lineweights*
- `invisible`: 0 if visible 1 if invisible, 0 is the default value
- `true_color`: true color as int value, see *ezdxf.colors*, has no default value
- `transparency`: transparency as int value, see *ezdxf.colors*, has no default value

A descriptor simplifies the attribute access through the *EntityQuery* container and has auto-completion support from IDEs:

```
result = msp.query(...)
# set attribute of all entities in result
result.layer = "MyLayer"
# delete attribute from all entities in result
del result.layer
# and for selector usage, see following section
assert len(result.layer == "MyLayer") == 1
```

Relational Selection Operators

The attribute selection by `__getitem__()` allows further selections by relational operators:

```
msp.add_line((0, 0), (1, 0), dxfattribs={"layer": "MyLayer})
lines = msp.query("LINE")
# select all entities on layer "MyLayer"
entities = lines["layer"] == "MyLayer"
assert len(entities) == 1

# or select all entities except the entities on layer "MyLayer"
entities = lines["layer"] != "MyLayer"
```

These operators work only with real DXF attributes, for instance the `rgb` attribute of graphical entities is not a real DXF attribute either the `vertices` of the LWPOLYLINE entity.

The selection by relational operators is case insensitive by default, because all names of DXF table entries are handled case insensitive. But if required the selection mode can be set to case sensitive:

```
lines = msp.query("LINE")
# use case sensitive selection: "MyLayer" != "MYLAYER"
lines.ignore_case = False
entities = lines["layer"] == "MYLAYER"
assert len(entities) == 0

# the result container has the default setting:
assert entities.ignore_case is True
```

Supported selection operators are:

- == equal “value”
- != not equal “value”
- < lower than “value”
- <= lower or equal than “value”
- > greater than “value”
- >= greater or equal than “value”

The relational operators <, >, <= and >= are not supported for vector-based attributes such as *center* or *insert* and raise a `TypeError`.

Note: These operators are selection operators and not logic operators, therefore the logic operators `and`, `or` and `not` are **not** applicable. The methods *union()*, *intersection()*, *difference()* and *symmetric_difference()*

can be used to combine selection. See section *Query Set Operators* and *Build Own Filters*.

Regular Expression Selection

The *EntityQuery.match()* method returns all entities where the selected DXF attribute matches the given regular expression. This methods work only on string based attributes, raises `TypeError` otherwise.

From here on I use only descriptors for attribute selection if possible.

```
msp.add_line((0, 0), (1, 0), dxfattribs={"layer": "Lay1"})
msp.add_line((0, 0), (1, 0), dxfattribs={"layer": "Lay2"})
lines = msp.query("LINE")

# select all entities at layers starting with "Lay",
# selection is also case insensitive by default:
assert len(lines.layer.match("^Lay.*")) == 2
```

Build Own Filters

The method *EntityQuery.filter* can be used to build operators for none-DXF attributes or for complex logic expressions.

Find all MTEXT entities in modelspace containing "SearchText". All *MText* entities have a `text` attribute, no need for a safety check:

```
mtext = msp.query("MTEXT").filter(lambda e: "SearchText" in e.text)
```

This filter checks the non-DXF attribute `rgb`. The filter has to check if the `rgb` attributes exist to avoid exceptions, because not all entities in modelspace may have the `rgb` attribute e.g. the `DXFTagStorage` entities which preserve unknown DXF entities:

```
result = msp.query().filter(
    lambda e: hasattr(e, "rgb") and e.rgb == (0, 0, 0)
)
```

Build 1-pass logic filters for complex queries, which would require otherwise multiple passes:

```
result = msp.query().filter(lambda e: e.dxf.color < 7 and e.dxf.layer == "0")
```

Combine filters for more complex operations. The first filter passes only valid entities and the second filter does the actual check:

```
def validator(entity):
    return True  # if entity is valid and has all required attributes

def check(entity):
    return True  # if entity passes the attribute checks

result = msp.query().filter(validator).filter(check)
```

Query Set Operators

The | operator or *EntityQuery.union()* returns a new *EntityQuery* with all entities from both queries. All entities are unique - no duplicates. This operator acts like the logical `or` operator.

```
entities = msp.query()
# select all entities with color < 2 or color > 7
result = (entities.color < 2 ) | (entities.color > 7)
```

The & operator or *EntityQuery.intersection()* returns a new *EntityQuery* with entities common to *self* and *other*. This operator acts like the logical `and` operator.

```
entities = msp.query()
# select all entities with color > 1 and color < 7
result = (entities.color > 1) & (entities.color < 7)
```

The – operator or *EntityQuery.difference()* returns a new *EntityQuery* with all entities from *self* that are not in *other*.

```
entities = msp.query()
# select all entities with color > 1 and not layer == "MyLayer"
result = (entities.color > 1) - (entities.layer != "MyLayer")
```

The ^ operator or *EntityQuery.symmetric_difference()* returns a new *EntityQuery* with entities in either *self* or *other* but not both.

```
entities = msp.query()
# select all entities with color > 1 or layer == "MyLayer", exclusive
# entities with color > 1 and layer == "MyLayer"
result = (entities.color > 1) ^ (entities.layer == "MyLayer")
```

The new() Function

ezdxf.query.**new**(*entities: Iterable[DXFEntity] | None = None*, *query: str = '*'*) → *EntityQuery*

Start a new query based on sequence *entities*. The *entities* argument has to be an iterable of *DXFEntity* or inherited objects and returns an *EntityQuery* object.

See also:

For usage of the groupby features see the tutorial: *Retrieve entities by groupby() function*

Groupby Function

ezdxf.groupby.**groupby**(*entities: Iterable[*DXFEntity*]*, *dxfattrib: str = ''*, *key: KeyFunc | None = None*) → dict[Hashable, list[*DXFEntity*]]

Groups a sequence of DXF entities by a DXF attribute like `'layer'`, returns a dict with *dxfattrib* values as key and a list of entities matching this *dxfattrib*. A *key* function can be used to combine some DXF attributes (e.g. layer and color) and should return a hashable data type like a tuple of strings, integers or floats, *key* function example:

```
def group_key(entity: DXFEntity):
    return entity.dxf.layer, entity.dxf.color
```

For not suitable DXF entities return `None` to exclude this entity, in this case it's not required, because *groupby()* catches `DXFAttributeError` exceptions to exclude entities, which do not provide layer and/or color attributes, automatically.

Result dict for *dxfattrib* = `'layer'` may look like this:

```
{
    '0': [ ... list of entities ],
    'ExampleLayer1': [ ... ],
    'ExampleLayer2': [ ... ],
    ...
}
```

Result dict for *key* = *group_key*, which returns a `(layer, color)` tuple, may look like this:

```
{
    ('0', 1): [ ... list of entities ],
    ('0', 3): [ ... ],
    ('0', 7): [ ... ],
    ('ExampleLayer1', 1): [ ... ],
    ('ExampleLayer1', 2): [ ... ],
    ('ExampleLayer1', 5): [ ... ],
    ('ExampleLayer2', 7): [ ... ],
    ...
}
```

All entity containers (modelspace, paperspace layouts and blocks) and the *EntityQuery* object have a dedicated *groupby()* method.

Parameters

- **entities** – sequence of DXF entities to group by a DXF attribute or a *key* function
- **dxfattrib** – grouping DXF attribute like `'layer'`
- **key** – key function, which accepts a `DXFEntity` as argument and returns a hashable grouping key or `None` to ignore this entity

6.9.7 Math

Core

Math core module: *ezdxf.math*

These are the core math functions and classes which should be imported from *ezdxf.math*.

Utility Functions

arc_angle_span_deg	Returns the counter-clockwise angle span from *start* to *end* in degrees.
arc_angle_span_rad	Returns the counter-clockwise angle span from *start* to *end* in radians.
arc_chord_length	Returns the chord length for an arc defined by *radius* and the sagitta.
arc_segment_count	Returns the count of required segments for the approximation of an arc for a given maximum sagitta.
area	Returns the area of a polygon, returns the projected area in the xy-plane for any vertices (z-axis will be ignored).
closest_point	Returns the closest point to a give *base* point.
ellipse_param_span	Returns the counter-clockwise params span of an elliptic arc from start- to end param.
has_matrix_2d_stretching	Returns `True` if matrix *m* performs a non-uniform xy-scaling.
has_matrix_3d_stretching	Returns `True` if matrix *m* performs a non-uniform xyz-scaling.
linspace	Return evenly spaced numbers over a specified interval, like numpy.linspace().
open_uniform_knot_vector	Returns an open (clamped) uniform knot vector for a B-spline of *order* and *count* control points.
required_knot_values	Returns the count of required knot-values for a B-spline of *order* and *count* control points.
uniform_knot_vector	Returns an uniform knot vector for a B-spline of *order* and *count* control points.
xround	Extended rounding function.

ezdxf.math.**closest_point**(*base:* UVec, *points: Iterable[*UVec*]*) → *Vec3*

Returns the closest point to a give *base* point.

Parameters

- **base** – base point as *Vec3* compatible object
- **points** – iterable of points as *Vec3* compatible object

ezdxf.math.**uniform_knot_vector**(*count: int*, *order: int*, *normalize=False*) → list[float]

Returns an uniform knot vector for a B-spline of *order* and *count* control points.

order = degree + 1

Parameters

- **count** – count of control points
- **order** – spline order
- **normalize** – normalize values in range [0, 1] if `True`

ezdxf.math.**open_uniform_knot_vector**(*count: int*, *order: int*, *normalize=False*) → list[float]

Returns an open (clamped) uniform knot vector for a B-spline of *order* and *count* control points.

order = degree + 1

Parameters

- **count** – count of control points
- **order** – spline order
- **normalize** – normalize values in range [0, 1] if `True`

ezdxf.math.**required_knot_values**(*count: int*, *order: int*) → int

Returns the count of required knot-values for a B-spline of *order* and *count* control points.

Parameters

- **count** – count of control points, in text-books referred as "n + 1"
- **order** – order of B-Spline, in text-books referred as "k"

Relationship:

"p" is the degree of the B-spline, text-book notation.

- k = p + 1
- 2 ≤ k ≤ n + 1

ezdxf.math.**xround**(*value: float*, *rounding: float = 0.*) → float

Extended rounding function.

The argument *rounding* defines the rounding limit:

0	remove fraction
0.1	round next to x.1, x.2, … x.0
0.25	round next to x.25, x.50, x.75 or x.00
0.5	round next to x.5 or x.0
1.0	round to a multiple of 1: remove fraction
2.0	round to a multiple of 2: xxx2, xxx4, xxx6 …
5.0	round to a multiple of 5: xxx5 or xxx0
10.0	round to a multiple of 10: xx10, xx20, …

Parameters

- **value** – float value to round
- **rounding** – rounding limit

ezdxf.math.**linspace**(*start: float*, *stop: float*, *num: int*, *endpoint=True*) → Iterable[float]

Return evenly spaced numbers over a specified interval, like numpy.linspace().

Returns *num* evenly spaced samples, calculated over the interval [start, stop]. The endpoint of the interval can optionally be excluded.

ezdxf.math.**area**(*vertices: Iterable[*UVec*]*) → float

Returns the area of a polygon, returns the projected area in the xy-plane for any vertices (z-axis will be ignored).

ezdxf.math.**arc_angle_span_deg**(*start: float*, *end: float*) → float

Returns the counter-clockwise angle span from *start* to *end* in degrees.

Returns the angle span in the range of [0, 360], 360 is a full circle. Full circle handling is a special case, because normalization of angles which describe a full circle would return 0 if treated as regular angles. e.g. (0, 360) → 360, (0, -360) → 360, (180, -180) → 360. Input angles with the same value always return 0 by definition: (0, 0) → 0, (-180, -180) → 0, (360, 360) → 0.

ezdxf.math.**arc_angle_span_rad**(*start: float*, *end: float*) → float

Returns the counter-clockwise angle span from *start* to *end* in radians.

Returns the angle span in the range of [0, 2π], 2π is a full circle. Full circle handling is a special case, because normalization of angles which describe a full circle would return 0 if treated as regular angles. e.g. (0, 2π) → 2π, (0, -2π) → 2π, (π, -π) → 2π. Input angles with the same value always return 0 by definition: (0, 0) → 0, (-π, -π) → 0, (2π, 2π) → 0.

ezdxf.math.**arc_segment_count**(*radius: float*, *angle: float*, *sagitta: float*) → int

Returns the count of required segments for the approximation of an arc for a given maximum sagitta.

Parameters

- **radius** – arc radius
- **angle** – angle span of the arc in radians
- **sagitta** – max. distance from the center of an arc segment to the center of its chord

ezdxf.math.**arc_chord_length**(*radius: float*, *sagitta: float*) → float

Returns the chord length for an arc defined by *radius* and the sagitta.

Parameters

- **radius** – arc radius
- **sagitta** – distance from the center of the arc to the center of its base

ezdxf.math.**ellipse_param_span**(*start_param: float*, *end_param: float*) → float

Returns the counter-clockwise params span of an elliptic arc from start- to end param.

Returns the param span in the range [0, 2π], 2π is a full ellipse. Full ellipse handling is a special case, because normalization of params which describe a full ellipse would return 0 if treated as regular params. e.g. (0, 2π) → 2π, (0, -2π) → 2π, (π, -π) → 2π. Input params with the same value always return 0 by definition: (0, 0) → 0, (-π, -π) → 0, (2π, 2π) → 0.

Alias to function: *ezdxf.math.arc_angle_span_rad()*

ezdxf.math.**has_matrix_2d_stretching**(*m:* Matrix44) → bool

Returns `True` if matrix *m* performs a non-uniform xy-scaling. Uniform scaling is not stretching in this context.

Does not check if the target system is a cartesian coordinate system, use the *Matrix44* property *is_cartesian* for that.

ezdxf.math.**has_matrix_3d_stretching**(*m:* Matrix44) → bool

Returns `True` if matrix *m* performs a non-uniform xyz-scaling. Uniform scaling is not stretching in this context.

Does not check if the target system is a cartesian coordinate system, use the *Matrix44* property *is_cartesian* for that.

Bulge Related Functions

arc_to_bulge	Returns bulge parameters from arc parameters.
bulge_3_points	Returns bulge value defined by three points.
bulge_center	Returns center of arc described by the given bulge parameters.
bulge_radius	Returns radius of arc defined by the given bulge parameters.
bulge_to_arc	Returns arc parameters from bulge parameters.
bulge_from_radius_and_chord	Returns the bulge value for the given arc radius and chord length.
bulge_from_arc_angle	Returns the bulge value for the given arc angle.

See also:

Description of the *Bulge value*.

ezdxf.math.**arc_to_bulge**(*center:* UVec, *start_angle: float*, *end_angle: float*, *radius: float*) → tuple[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec2*, float]

Returns bulge parameters from arc parameters.

Parameters

- **center** – circle center point as *Vec2* compatible object
- **start_angle** – start angle in radians
- **end_angle** – end angle in radians
- **radius** – circle radius

Returns
(start_point, end_point, bulge)

Return type
tuple

ezdxf.math.**bulge_3_points**(*start_point:* UVec, *end_point:* UVec, *point:* UVec) → float

Returns bulge value defined by three points.

Based on 3-Points to Bulge by Lee Mac.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **point** – arbitrary point as *Vec2* compatible object

ezdxf.math.**bulge_center**(*start_point:* UVec, *end_point:* UVec, *bulge: float*) → *Vec2*

Returns center of arc described by the given bulge parameters.

Based on Bulge Center by Lee Mac.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **bulge** – bulge value as float

ezdxf.math.**bulge_radius**(*start_point:* UVec, *end_point:* UVec, *bulge: float*) → float

Returns radius of arc defined by the given bulge parameters.

Based on Bulge Radius by Lee Mac

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **bulge** – bulge value

ezdxf.math.**bulge_to_arc**(*start_point:* UVec, *end_point:* UVec, *bulge: float*) → tuple[*ezdxf.math._vector.Vec2*, float, float, float]

Returns arc parameters from bulge parameters.

The arcs defined by bulge values of *LWPolyline* and 2D *Polyline* entities start at the vertex which includes the bulge value and ends at the following vertex.

Based on Bulge to Arc by Lee Mac.

Parameters

- **start_point** – start vertex as *Vec2* compatible object
- **end_point** – end vertex as *Vec2* compatible object
- **bulge** – bulge value

Returns
(center, start_angle, end_angle, radius)

Return type
Tuple

ezdxf.math.**bulge_from_radius_and_chord**(*radius: float*, *chord: float*) → float

Returns the bulge value for the given arc radius and chord length. Returns 0 if the radius is zero or the radius is too small for the given chord length to create an arc.

Parameters

- **radius** – arc radius
- **chord** – chord length

ezdxf.math.**bulge_from_arc_angle**(*angle: float*) → float

Returns the bulge value for the given arc angle.

Parameters
angle – arc angle in radians

2D Graphic Functions

convex_hull_2d	Returns the 2D convex hull of given *points*.
distance_point_line_2d	Returns the normal distance from *point* to 2D line defined by *start*- and *end* point.
intersect_polylines_2d	Returns the intersection points for two polylines as list of *Vec2* objects, the list is empty if no intersection points exist.
intersection_line_line_2d	Compute the intersection of two lines in the xy-plane.
is_convex_polygon_2d	Returns `True` if the 2D *polygon* is convex.
is_point_in_polygon_2d	Test if *point* is inside *polygon*.
is_point_left_of_line	Returns `True` if *point* is "left of line" defined by *start*- and *end* point, a colinear point is also "left of line" if argument *colinear* is `True`.
is_point_on_line_2d	Returns `True` if *point* is on *line*.
offset_vertices_2d	Yields vertices of the offset line to the shape defined by *vertices*.
point_to_line_relation	Returns `-1` if *point* is left *line*, `+1` if *point* is right of *line* and `0` if *point* is on the *line*.
rytz_axis_construction	The Rytz's axis construction is a basic method of descriptive Geometry to find the axes, the semi-major axis and semi-minor axis, starting from two conjugated half-diameters.

ezdxf.math.**convex_hull_2d**(*points: Iterable[*UVec*]*) → list[*ezdxf.math._vector.Vec2*]

Returns the 2D convex hull of given *points*.

Returns a closed polyline, first vertex is equal to the last vertex.

Parameters

points – iterable of points, z-axis is ignored

ezdxf.math.**distance_point_line_2d**(*point:* Vec2, *start:* Vec2, *end:* Vec2) → float

Returns the normal distance from *point* to 2D line defined by *start*- and *end* point.

ezdxf.math.**intersect_polylines_2d**(*p1: Sequence[*Vec2*]*, *p2: Sequence[*Vec2*]*, *abs_tol=1e-10*) → list[*ezdxf.math._vector.Vec2*]

Returns the intersection points for two polylines as list of *Vec2* objects, the list is empty if no intersection points exist. Does not return self intersection points of *p1* or *p2*. Duplicate intersection points are removed from the result list, but the list does not have a particular order! You can sort the result list by `result.sort()` to introduce an order.

Parameters

- **p1** – first polyline as sequence of *Vec2* objects
- **p2** – second polyline as sequence of *Vec2* objects
- **abs_tol** – absolute tolerance for comparisons

ezdxf.math.**intersection_line_line_2d**(*line1: Sequence[*Vec2*]*, *line2: Sequence[*Vec2*]*, *virtual=True*, *abs_tol=TOLERANCE*) → *Vec2* | None

Compute the intersection of two lines in the xy-plane.

Parameters

- **line1** – start- and end point of first line to test e.g. ((x1, y1), (x2, y2)).

- **line2** – start- and end point of second line to test e.g. ((x3, y3), (x4, y4)).
- **virtual** – `True` returns any intersection point, `False` returns only real intersection points.
- **abs_tol** – tolerance for intersection test.

Returns
`None` if there is no intersection point (parallel lines) or intersection point as *Vec2*

`ezdxf.math.`**is_convex_polygon_2d**(*polygon: list[*ezdxf.math._vector.Vec2*]*, *, *strict=False*, *epsilon=1e-6*) → bool

Returns `True` if the 2D *polygon* is convex. This function works with open and closed polygons and clockwise or counter-clockwise vertex orientation. Coincident vertices will always be skipped and if argument *strict* is `True`, polygons with collinear vertices are not considered as convex.

This solution works only for simple non-self-intersecting polygons!

`ezdxf.math.`**is_point_in_polygon_2d**(*point:* Vec2, *polygon: Sequence[*Vec2*]*, *abs_tol=TOLERANCE*) → int

Test if *point* is inside *polygon*. Returns `-1` (for outside) if the polygon is degenerated, no exception will be raised.

Parameters

- **point** – 2D point to test as *Vec2*
- **polygon** – sequence of 2D points as *Vec2*
- **abs_tol** – tolerance for distance check

Returns
`+1` for inside, `0` for on boundary line, `-1` for outside

`ezdxf.math.`**is_point_left_of_line**(*point:* Vec2, *start:* Vec2, *end:* Vec2, *colinear=False*) → bool

Returns `True` if *point* is "left of line" defined by *start*- and *end* point, a colinear point is also "left of line" if argument *colinear* is `True`.

Parameters

- **point** – 2D point to test as *Vec2*
- **start** – line definition point as *Vec2*
- **end** – line definition point as *Vec2*
- **colinear** – a colinear point is also "left of line" if `True`

`ezdxf.math.`**is_point_on_line_2d**(*point:* Vec2, *start:* Vec2, *end:* Vec2, *ray=True*, *abs_tol=TOLERANCE*) → bool

Returns `True` if *point* is on *line*.

Parameters

- **point** – 2D point to test as *Vec2*
- **start** – line definition point as *Vec2*
- **end** – line definition point as *Vec2*
- **ray** – if `True` point has to be on the infinite ray, if `False` point has to be on the line segment
- **abs_tol** – tolerance for on the line test

ezdxf.math.**offset_vertices_2d**(*vertices: Iterable[*UVec*]*, *offset: float*, *closed: bool = False*) → Iterable[*Vec2*]

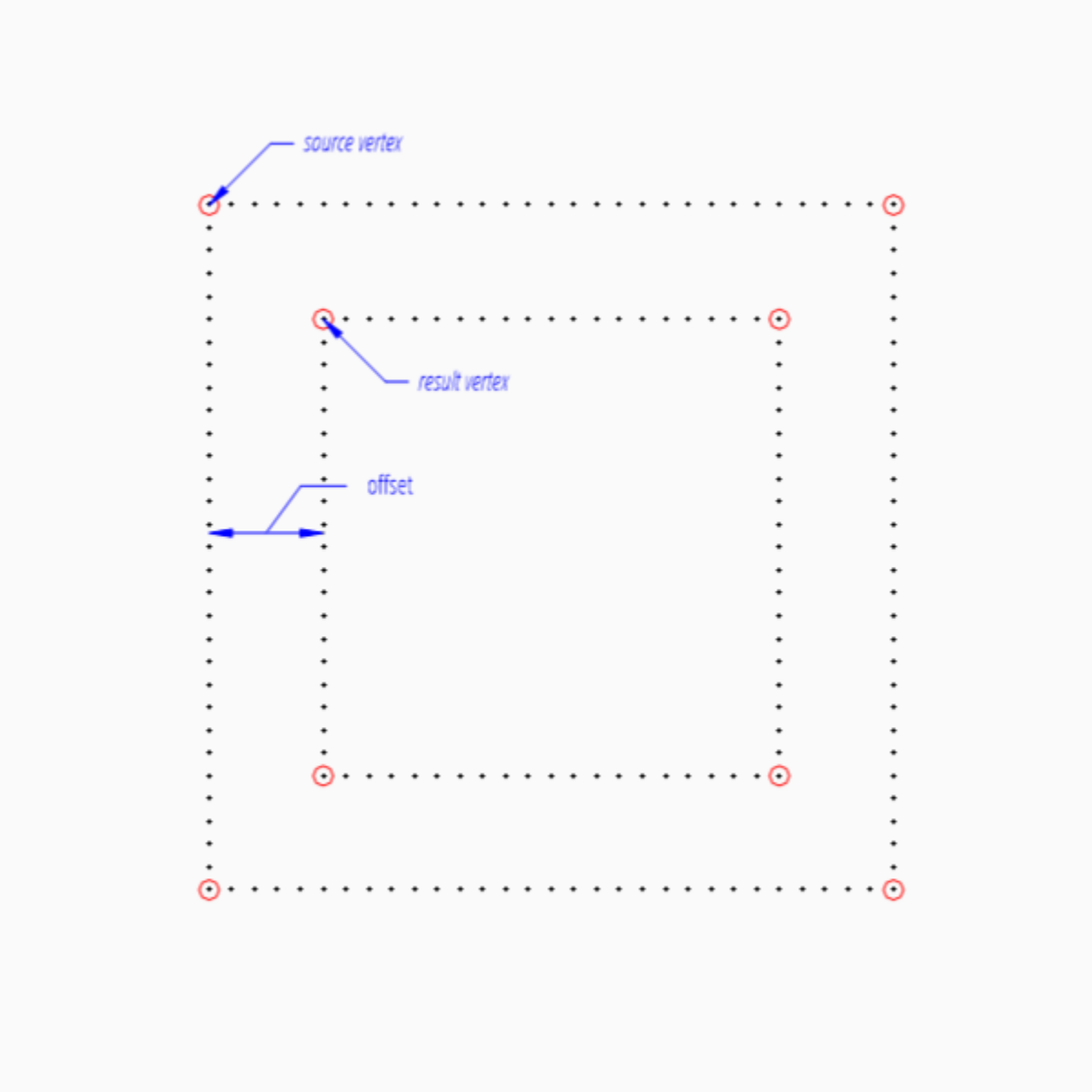
Yields vertices of the offset line to the shape defined by *vertices*. The source shape consist of straight segments and is located in the xy-plane, the z-axis of input vertices is ignored. Takes closed shapes into account if argument *closed* is `True`, which yields intersection of first and last offset segment as first vertex for a closed shape. For closed shapes the first and last vertex can be equal, else an implicit closing segment from last to first vertex is added. A shape with equal first and last vertex is not handled automatically as closed shape.

Warning: Adjacent collinear segments in *opposite* directions, same as a turn by 180 degree (U-turn), leads to unexpected results.

Parameters

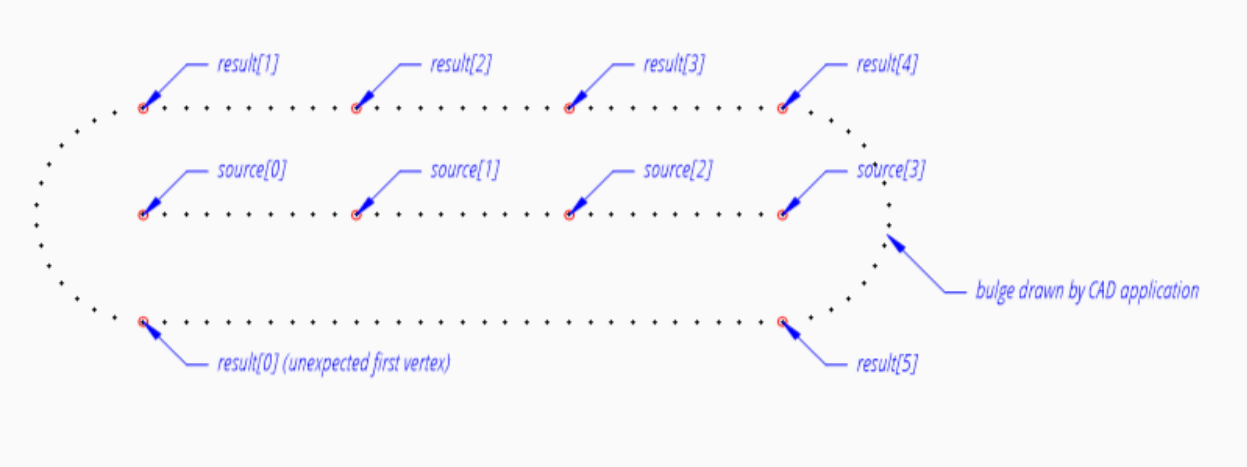
- **vertices** – source shape defined by vertices
- **offset** – line offset perpendicular to direction of shape segments defined by vertices order, offset > `0` is 'left' of line segment, offset < `0` is 'right' of line segment
- **closed** – `True` to handle as closed shape

```
source = [(0, 0), (3, 0), (3, 3), (0, 3)]
result = list(offset_vertices_2d(source, offset=0.5, closed=True))
```



Example for a closed collinear shape, which creates 2 additional vertices and the first one has an unexpected location:

```
source = [(0, 0), (0, 1), (0, 2), (0, 3)]
result = list(offset_vertices_2d(source, offset=0.5, closed=True))
```



ezdxf.math.**point_to_line_relation**(*point:* Vec2, *start:* Vec2, *end:* Vec2, *abs_tol=TOLERANCE*) → int

Returns `-1` if *point* is left *line*, `+1` if *point* is right of *line* and `0` if *point* is on the *line*. The *line* is defined by two vertices given as arguments *start* and *end*.

Parameters

- **point** – 2D point to test as *Vec2*
- **start** – line definition point as *Vec2*
- **end** – line definition point as *Vec2*
- **abs_tol** – tolerance for minimum distance to line

ezdxf.math.**rytz_axis_construction**(*d1:* Vec3, *d2:* Vec3) → tuple[*ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, float]

The Rytz's axis construction is a basic method of descriptive Geometry to find the axes, the semi-major axis and semi-minor axis, starting from two conjugated half-diameters.

Source: Wikipedia

Given conjugated diameter *d1* is the vector from center C to point P and the given conjugated diameter *d2* is the vector from center C to point Q. Center of ellipse is always `(0, 0, 0)`. This algorithm works for 2D/3D vectors.

Parameters

- **d1** – conjugated semi-major axis as *Vec3*
- **d2** – conjugated semi-minor axis as *Vec3*

Returns

Tuple of (major axis, minor axis, ratio)

3D Graphic Functions

basic_transformation	Returns a combined transformation matrix for translation, scaling and rotation about the z-axis.
best_fit_normal	Returns the "best fit" normal for a plane defined by three or more vertices.
bezier_to_bspline	Convert multiple quadratic or cubic Bèzier curves into a single cubic B-spline.
closed_uniform_bspline	Creates a closed uniform (periodic) B-spline curve (open curve).
cubic_bezier_bbox	Returns the *BoundingBox* of a cubic Bézier curve of type *Bezier4P*.
cubic_bezier_from_3p	Returns a cubic Bèzier curve *Bezier4P* from three points.
cubic_bezier_from_arc	Returns an approximation for a circular 2D arc by multiple cubic Bézier-curves.
cubic_bezier_from_ellipse	Returns an approximation for an elliptic arc by multiple cubic Bézier-curves.
cubic_bezier_interpolation	Returns an interpolation curve for given data *points* as multiple cubic Bézier-curves.
distance_point_line_3d	Returns the normal distance from a *point* to a 3D line.
estimate_end_tangent_magnitude	Estimate tangent magnitude of start- and end tangents.
estimate_tangents	Estimate tangents for curve defined by given fit points.
fit_points_to_cad_cv	Returns a cubic *BSpline* from fit points as close as possible to common CAD applications like BricsCAD.
fit_points_to_cubic_bezier	Returns a cubic *BSpline* from fit points **without** end tangents.
global_bspline_interpolation	B-spline interpolation by the Global Curve Interpolation.
have_bezier_curves_g1_continuity	Return `True` if the given adjacent Bézier curves have G1 continuity.
intersect_polylines_3d	Returns the intersection points for two polylines as list of *Vec3* objects, the list is empty if no intersection points exist.
intersection_line_line_3d	Returns the intersection point of two 3D lines, returns `None` if lines do not intersect.
intersection_line_polygon_3d	Returns the intersection point of the 3D line form *start* to *end* and the given *polygon*.
intersection_ray_polygon_3d	Returns the intersection point of the infinite 3D ray defined by *origin* and the *direction* vector and the given *polygon*.
intersection_ray_ray_3d	Calculate intersection of two 3D rays, returns a 0-tuple for parallel rays, a 1-tuple for intersecting rays and a 2-tuple for not intersecting and not parallel rays with points of the closest approach on each ray.
is_planar_face	Returns `True` if sequence of vectors is a planar face.
linear_vertex_spacing	Returns *count* evenly spaced vertices from *start* to *end*.
local_cubic_bspline_interpolation	B-spline interpolation by 'Local Cubic Curve Interpolation', which creates B-spline from fit points and estimated tangent direction at start-, end- and passing points.
normal_vector_3p	Returns normal vector for 3 points, which is the normalized cross product for: `a->b x a->c`.

continues on next page

Table 2 – continued from previous page

open_uniform_bspline	Creates an open uniform (periodic) B-spline curve (open curve).
quadratic_bezier_bbox	Returns the *BoundingBox* of a quadratic Bézier curve of type *Bezier3P*.
quadratic_bezier_from_3p	Returns a quadratic Bèzier curve *Bezier3P* from three points.
quadratic_to_cubic_bezier	Convert quadratic Bèzier curves (*ezdxf.math.Bezier3P*) into cubic Bèzier curves (*ezdxf.math.Bezier4P*).
rational_bspline_from_arc	Returns a rational B-splines for a circular 2D arc.
rational_bspline_from_ellipse	Returns a rational B-splines for an elliptic arc.
safe_normal_vector	Safe function to detect the normal vector for a face or polygon defined by 3 or more *vertices*.
spherical_envelope	Calculate the spherical envelope for the given points.
split_bezier	Split a Bèzier curve at parameter *t*.
split_polygon_by_plane	Split a convex *polygon* by the given *plane*.
subdivide_face	Subdivides faces by subdividing edges and adding a center vertex.
subdivide_ngons	Subdivides faces into triangles by adding a center vertex.

See also:

The free online book 3D Math Primer for Graphics and Game Development is a very good resource for learning vector math and other graphic related topics, it is easy to read for beginners and especially targeted to programmers.

ezdxf.math.**basic_transformation**(*move:* UVec = *(0, 0, 0)*, *scale:* UVec = *(1, 1, 1)*, *z_rotation: float = 0*) → *Matrix44*

Returns a combined transformation matrix for translation, scaling and rotation about the z-axis.

Parameters

- **move** – translation vector
- **scale** – x-, y- and z-axis scaling as float triplet, e.g. (2, 2, 1)
- **z_rotation** – rotation angle about the z-axis in radians

ezdxf.math.**best_fit_normal**(*vertices: Iterable[*UVec*]*) → *Vec3*

Returns the "best fit" normal for a plane defined by three or more vertices. This function tolerates imperfect plane vertices. Safe function to detect the extrusion vector of flat arbitrary polygons.

ezdxf.math.**bezier_to_bspline**(*curves: Iterable[*Bezier3P | Bezier4P*]*) → *BSpline*

Convert multiple quadratic or cubic Bèzier curves into a single cubic B-spline.

For good results the curves must be lined up seamlessly, i.e. the starting point of the following curve must be the same as the end point of the previous curve. G1 continuity or better at the connection points of the Bézier curves is required to get best results.

ezdxf.math.**closed_uniform_bspline**(*control_points: Iterable[*UVec*]*, *order: int = 4*, *weights: Iterable[float] | None = None*) → *BSpline*

Creates a closed uniform (periodic) B-spline curve (open curve).

This B-spline does not pass any of the control points.

Parameters

- **control_points** – iterable of control points as *Vec3* compatible objects

- **order** – spline order (degree + 1)
- **weights** – iterable of weight values

ezdxf.math.**cubic_bezier_bbox**(*curve:* Bezier4P, *, *abs_tol=1e-12*) → *BoundingBox*

Returns the *BoundingBox* of a cubic Bézier curve of type *Bezier4P*.

ezdxf.math.**cubic_bezier_from_3p**(*p1:* UVec, *p2:* UVec, *p3:* UVec) → *Bezier4P*

Returns a cubic Bèzier curve *Bezier4P* from three points. The curve starts at *p1*, goes through *p2* and ends at *p3*. (source: pomax-2)

ezdxf.math.**cubic_bezier_from_arc**(*center:* UVec = *(0, 0, 0)*, *radius: float = 1*, *start_angle: float = 0*, *end_angle: float = 360*, *segments: int = 1*) → Iterable[*Bezier4P*]

Returns an approximation for a circular 2D arc by multiple cubic Bézier-curves.

Parameters

- **center** – circle center as *Vec3* compatible object
- **radius** – circle radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **segments** – count of Bèzier-curve segments, at least one segment for each quarter (90 deg), 1 for as few as possible.

ezdxf.math.**cubic_bezier_from_ellipse**(*ellipse:* ConstructionEllipse, *segments: int = 1*) → Iterable[*Bezier4P*]

Returns an approximation for an elliptic arc by multiple cubic Bézier-curves.

Parameters

- **ellipse** – ellipse parameters as *ConstructionEllipse* object
- **segments** – count of Bèzier-curve segments, at least one segment for each quarter (π/2), 1 for as few as possible.

ezdxf.math.**cubic_bezier_interpolation**(*points: Iterable[*UVec*]*) → Iterable[*Bezier4P*]

Returns an interpolation curve for given data *points* as multiple cubic Bézier-curves. Returns n-1 cubic Bézier-curves for n given data points, curve i goes from point[i] to point[i+1].

Parameters

points – data points

ezdxf.math.**distance_point_line_3d**(*point:* Vec3, *start:* Vec3, *end:* Vec3) → float

Returns the normal distance from a *point* to a 3D line.

Parameters

- **point** – point to test
- **start** – start point of the 3D line
- **end** – end point of the 3D line

ezdxf.math.**estimate_end_tangent_magnitude**(*points: list[*ezdxf.math._vector.Vec3*]*, *method: str = 'chord'*) → tuple[float, float]

Estimate tangent magnitude of start- and end tangents.

Available estimation methods:

- "chord": total chord length, curve approximation by straight segments

- "arc": total arc length, curve approximation by arcs
- "bezier-n": total length from cubic bezier curve approximation, n segments per section

Parameters

- **points** – start-, end- and passing points of curve
- **method** – tangent magnitude estimation method

ezdxf.math.**estimate_tangents**(*points: list[*ezdxf.math._vector.Vec3*]*, *method: str = '5-points'*, *normalize=True*) → list[*ezdxf.math._vector.Vec3*]

Estimate tangents for curve defined by given fit points. Calculated tangents are normalized (unit-vectors).

Available tangent estimation methods:

- "3-points": 3 point interpolation
- "5-points": 5 point interpolation
- "bezier": tangents from an interpolated cubic bezier curve
- "diff": finite difference

Parameters

- **points** – start-, end- and passing points of curve
- **method** – tangent estimation method
- **normalize** – normalize tangents if `True`

Returns
tangents as list of *Vec3* objects

ezdxf.math.**fit_points_to_cad_cv**(*fit_points: Iterable[*UVec*]*, *tangents: Iterable[*UVec*] | None = None*) → *BSpline*

Returns a cubic *BSpline* from fit points as close as possible to common CAD applications like BricsCAD.

There exist infinite numerical correct solution for this setup, but some facts are known:

- CAD applications use the global curve interpolation with start- and end derivatives if the end tangents are defined otherwise the equation system will be completed by setting the second derivatives of the start and end point to 0, for more information read this answer on stackoverflow: https://stackoverflow.com/a/74863330/6162864
- The degree of the B-spline is always 3 regardless which degree is stored in the SPLINE entity, this is only valid for B-splines defined by fit points
- Knot parametrization method is "chord"
- Knot distribution is "natural"

Parameters

- **fit_points** – points the spline is passing through
- **tangents** – start- and end tangent, default is autodetect

ezdxf.math.**fit_points_to_cubic_bezier**(*fit_points: Iterable[UVec]*) → *BSpline*

Returns a cubic *BSpline* from fit points **without** end tangents.

This function uses the cubic Bèzier interpolation to create multiple Bèzier curves and combine them into a single B-spline, this works for short simple splines better than the *fit_points_to_cad_cv()*, but is worse for longer and more complex splines.

Parameters

fit_points – points the spline is passing through

ezdxf.math.**global_bspline_interpolation**(*fit_points: Iterable[UVec]*, *degree: int = 3*, *tangents: Iterable[UVec] | None = None*, *method: str = 'chord'*) → *BSpline*

B-spline interpolation by the Global Curve Interpolation. Given are the fit points and the degree of the B-spline. The function provides 3 methods for generating the parameter vector t:

- "uniform": creates a uniform t vector, from 0 to 1 evenly spaced, see uniform method
- "chord", "distance": creates a t vector with values proportional to the fit point distances, see chord length method
- "centripetal", "sqrt_chord": creates a t vector with values proportional to the fit point sqrt(distances), see centripetal method
- "arc": creates a t vector with values proportional to the arc length between fit points.

It is possible to constraint the curve by tangents, by start- and end tangent if only two tangents are given or by one tangent for each fit point.

If tangents are given, they represent 1st derivatives and should be scaled if they are unit vectors, if only start- and end tangents given the function *estimate_end_tangent_magnitude()* helps with an educated guess, if all tangents are given, scaling by chord length is a reasonable choice (Piegl & Tiller).

Parameters

- **fit_points** – fit points of B-spline, as list of *Vec3* compatible objects
- **tangents** – if only two vectors are given, take the first and the last vector as start- and end tangent constraints or if for all fit points a tangent is given use all tangents as interpolation constraints (optional)
- **degree** – degree of B-spline
- **method** – calculation method for parameter vector t

Returns

BSpline

ezdxf.math.**have_bezier_curves_g1_continuity**(*b1: Bezier3P | Bezier4P*, *b2: Bezier3P | Bezier4P*, *g1_tol: float = 1e-4*) → bool

Return `True` if the given adjacent Bézier curves have G1 continuity.

ezdxf.math.**intersect_polylines_3d**(*p1: Sequence[Vec3]*, *p2: Sequence[Vec3]*, *abs_tol=1e-10*) → list[*ezdxf.math._vector.Vec3*]

Returns the intersection points for two polylines as list of *Vec3* objects, the list is empty if no intersection points exist. Does not return self intersection points of *p1* or *p2*. Duplicate intersection points are removed from the result list, but the list does not have a particular order! You can sort the result list by `result.sort()` to introduce an order.

Parameters

- **p1** – first polyline as sequence of *Vec3* objects

- **p2** – second polyline as sequence of *Vec3* objects
- **abs_tol** – absolute tolerance for comparisons

ezdxf.math.**intersection_line_line_3d**(*line1: Sequence[Vec3]*, *line2: Sequence[Vec3]*, *virtual: bool = True*, *abs_tol: float = 1e-10*) → *Vec3* | None

Returns the intersection point of two 3D lines, returns `None` if lines do not intersect.

Parameters

- **line1** – first line as tuple of two points as *Vec3* objects
- **line2** – second line as tuple of two points as *Vec3* objects
- **virtual** – `True` returns any intersection point, `False` returns only real intersection points
- **abs_tol** – absolute tolerance for comparisons

ezdxf.math.**intersection_line_polygon_3d**(*start: Vec3*, *end: Vec3*, *polygon: Iterable[Vec3]*, ***, *coplanar=True*, *boundary=True*, *abs_tol=PLANE_EPSILON*) → *Vec3* | None

Returns the intersection point of the 3D line form *start* to *end* and the given *polygon*.

Parameters

- **start** – start point of 3D line as *Vec3*
- **end** – end point of 3D line as *Vec3*
- **polygon** – 3D polygon as iterable of *Vec3*
- **coplanar** – if `True` a coplanar start- or end point as intersection point is valid
- **boundary** – if `True` an intersection point at the polygon boundary line is valid
- **abs_tol** – absolute tolerance for comparisons

ezdxf.math.**intersection_ray_polygon_3d**(*origin: Vec3*, *direction: Vec3*, *polygon: Iterable[Vec3]*, ***, *boundary=True*, *abs_tol=PLANE_EPSILON*) → *Vec3* | None

Returns the intersection point of the infinite 3D ray defined by *origin* and the *direction* vector and the given *polygon*.

Parameters

- **origin** – origin point of the 3D ray as *Vec3*
- **direction** – direction vector of the 3D ray as *Vec3*
- **polygon** – 3D polygon as iterable of *Vec3*
- **boundary** – if `True` intersection points at the polygon boundary line are valid
- **abs_tol** – absolute tolerance for comparisons

ezdxf.math.**intersection_ray_ray_3d**(*ray1: Sequence[Vec3]*, *ray2: Sequence[Vec3]*, *abs_tol=TOLERANCE*) → Sequence[*Vec3*]

Calculate intersection of two 3D rays, returns a 0-tuple for parallel rays, a 1-tuple for intersecting rays and a 2-tuple for not intersecting and not parallel rays with points of the closest approach on each ray.

Parameters

- **ray1** – first ray as tuple of two points as *Vec3* objects
- **ray2** – second ray as tuple of two points as *Vec3* objects
- **abs_tol** – absolute tolerance for comparisons

ezdxf.math.**is_planar_face**(*face: Sequence[*Vec3*]*, *abs_tol=1e-9*) → bool

Returns `True` if sequence of vectors is a planar face.

Parameters

- **face** – sequence of *Vec3* objects
- **abs_tol** – tolerance for normals check

ezdxf.math.**linear_vertex_spacing**(*start:* Vec3, *end:* Vec3, *count: int*) → list[*ezdxf.math._vector.Vec3*]

Returns *count* evenly spaced vertices from *start* to *end*.

ezdxf.math.**local_cubic_bspline_interpolation**(*fit_points: Iterable[*UVec*]*, *method: str = '5-points'*, *tangents: Iterable[*UVec*] | None = None*) → *BSpline*

B-spline interpolation by 'Local Cubic Curve Interpolation', which creates B-spline from fit points and estimated tangent direction at start-, end- and passing points.

Source: Piegl & Tiller: "The NURBS Book" - chapter 9.3.4

Available tangent estimation methods:

- "3-points": 3 point interpolation
- "5-points": 5 point interpolation
- "bezier": cubic bezier curve interpolation
- "diff": finite difference

or pass pre-calculated tangents, which overrides tangent estimation.

Parameters

- **fit_points** – all B-spline fit points as *Vec3* compatible objects
- **method** – tangent estimation method
- **tangents** – tangents as *Vec3* compatible objects (optional)

Returns
BSpline

ezdxf.math.**normal_vector_3p**(*a:* Vec3, *b:* Vec3, *c:* Vec3) → *Vec3*

Returns normal vector for 3 points, which is the normalized cross product for: `a->b x a->c`.

ezdxf.math.**open_uniform_bspline**(*control_points: Iterable[*UVec*]*, *order: int = 4*, *weights: Iterable[float] | None = None*) → *BSpline*

Creates an open uniform (periodic) B-spline curve (open curve).

This is an unclamped curve, which means the curve passes none of the control points.

Parameters

- **control_points** – iterable of control points as *Vec3* compatible objects
- **order** – spline order (degree + 1)
- **weights** – iterable of weight values

ezdxf.math.**quadratic_bezier_bbox**(*curve:* Bezier3P, *, *abs_tol=1e-12*) → *BoundingBox*

Returns the *BoundingBox* of a quadratic Bézier curve of type *Bezier3P*.

ezdxf.math.**quadratic_bezier_from_3p**(*p1:* UVec, *p2:* UVec, *p3:* UVec) → *Bezier3P*

Returns a quadratic Bèzier curve *Bezier3P* from three points. The curve starts at *p1*, goes through *p2* and ends at *p3*. (source: pomax-2)

ezdxf.math.**quadratic_to_cubic_bezier**(*curve:* Bezier3P) → *Bezier4P*

Convert quadratic Bèzier curves (*ezdxf.math.Bezier3P*) into cubic Bèzier curves (*ezdxf.math.Bezier4P*).

ezdxf.math.**rational_bspline_from_arc**(*center:* Vec3 = *(0, 0)*, *radius: float = 1*, *start_angle: float = 0*, *end_angle: float = 360*, *segments: int = 1*) → *BSpline*

Returns a rational B-splines for a circular 2D arc.

Parameters

- **center** – circle center as *Vec3* compatible object
- **radius** – circle radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **segments** – count of spline segments, at least one segment for each quarter (90 deg), default is 1, for as few as needed.

ezdxf.math.**rational_bspline_from_ellipse**(*ellipse:* ConstructionEllipse, *segments: int = 1*) → *BSpline*

Returns a rational B-splines for an elliptic arc.

Parameters

- **ellipse** – ellipse parameters as *ConstructionEllipse* object
- **segments** – count of spline segments, at least one segment for each quarter (π/2), default is 1, for as few as needed.

ezdxf.math.**safe_normal_vector**(*vertices: Sequence[*Vec3*]*) → *Vec3*

Safe function to detect the normal vector for a face or polygon defined by 3 or more *vertices*.

ezdxf.math.**spherical_envelope**(*points: Sequence[*UVec*]*) → tuple[*ezdxf.math._vector.Vec3*, float]

Calculate the spherical envelope for the given points. Returns the centroid (a.k.a. geometric center) and the radius of the enclosing sphere.

Note: The result does not represent the minimal bounding sphere!

---

ezdxf.math.**split_bezier**(*control_points: Sequence[T]*, *t: float*) → tuple[list[T], list[T]]

Split a Bèzier curve at parameter *t*.

Returns the control points for two new Bèzier curves of the same degree and type as the input curve. (source: pomax-1)

Parameters

- **control_points** – of the Bèzier curve as *Vec2* or *Vec3* objects. Requires 3 points for a quadratic curve, 4 points for a cubic curve , …
- **t** – parameter where to split the curve in the range [0, 1]

ezdxf.math.**split_polygon_by_plane**(*polygon: Iterable[*Vec3*]*, *plane:* Plane, *, *coplanar=True*, *abs_tol=PLANE_EPSILON*) → tuple[Sequence[*ezdxf.math._vector.Vec3*], Sequence[*ezdxf.math._vector.Vec3*]]

Split a convex *polygon* by the given *plane*.

Returns a tuple of front- and back vertices (front, back). Returns also coplanar polygons if the argument *coplanar* is `True`, the coplanar vertices goes into either front or back depending on their orientation with respect to this plane.

ezdxf.math.**subdivide_face**(*face: Sequence[*Vec2 | Vec3*]*, *quads: bool = True*) → Iterable[tuple[*ezdxf.math._vector.Vec3*, ...]]

Subdivides faces by subdividing edges and adding a center vertex.

Parameters

- **face** – a sequence of vertices, *Vec2* and *Vec3* objects supported.
- **quads** – create quad faces if `True` else create triangles

ezdxf.math.**subdivide_ngons**(*faces: Iterable[Sequence[*Vec2 | Vec3*]]*, *max_vertex_count=4*) → Iterable[Sequence[*Vec3*]]

Subdivides faces into triangles by adding a center vertex.

Parameters

- **faces** – iterable of faces as sequence of *Vec2* and *Vec3* objects
- **max_vertex_count** – subdivide only ngons with more vertices

Transformation Classes

Matrix44	An optimized 4x4 transformation matrix.
OCS	Establish an *OCS* for a given extrusion vector.
UCS	Establish a user coordinate system (*UCS*).

OCS Class

class ezdxf.math.**OCS**(*extrusion:* UVec = *Z_AXIS*)

Establish an *OCS* for a given extrusion vector.

Parameters
extrusion – extrusion vector.

ux

x-axis unit vector

uy

y-axis unit vector

uz

z-axis unit vector

from_wcs(*point:* UVec) → *UVec*

Returns OCS vector for WCS *point*.

points_from_wcs (*points: Iterable[*UVec*]*) → Iterable[*UVec*]

Returns iterable of OCS vectors from WCS *points*.

to_wcs (*point:* UVec) → *UVec*

Returns WCS vector for OCS *point*.

points_to_wcs (*points: Iterable[*UVec*]*) → Iterable[*UVec*]

Returns iterable of WCS vectors for OCS *points*.

render_axis (*layout:* BaseLayout, *length: float = 1*, *colors: RGB = (1, 3, 5)*) → None

Render axis as 3D lines into a *layout*.

UCS Class

class ezdxf.math.**UCS** (*origin:* UVec = *(0, 0, 0)*, *ux:* UVec | *None* = *None*, *uy:* UVec | *None* = *None*, *uz:* UVec | *None* = *None*)

Establish a user coordinate system (*UCS*). The UCS is defined by the origin and two unit vectors for the x-, y- or z-axis, all axis in *WCS*. The missing axis is the cross product of the given axis.

If x- and y-axis are `None`: ux = `(1, 0, 0)`, uy = `(0, 1, 0)`, uz = `(0, 0, 1)`.

Unit vectors don't have to be normalized, normalization is done at initialization, this is also the reason why scaling gets lost by copying or rotating.

Parameters

- **origin** – defines the UCS origin in world coordinates
- **ux** – defines the UCS x-axis as vector in *WCS*
- **uy** – defines the UCS y-axis as vector in *WCS*
- **uz** – defines the UCS z-axis as vector in *WCS*

ux

x-axis unit vector

uy

y-axis unit vector

uz

z-axis unit vector

is_cartesian

Returns `True` if cartesian coordinate system.

copy () → *UCS*

Returns a copy of this UCS.

to_wcs (*point:* Vec3) → *Vec3*

Returns WCS point for UCS *point*.

points_to_wcs (*points: Iterable[*Vec3*]*) → Iterable[*Vec3*]

Returns iterable of WCS vectors for UCS *points*.

direction_to_wcs (*vector:* Vec3) → *Vec3*

Returns WCS direction for UCS *vector* without origin adjustment.

from_wcs(*point:* Vec3) → *Vec3*

Returns UCS point for WCS *point*.

points_from_wcs(*points: Iterable[*Vec3*]*) → Iterable[*Vec3*]

Returns iterable of UCS vectors from WCS *points*.

direction_from_wcs(*vector:* Vec3) → *Vec3*

Returns UCS vector for WCS *vector* without origin adjustment.

to_ocs(*point:* Vec3) → *Vec3*

Returns OCS vector for UCS *point*.

The *OCS* is defined by the z-axis of the *UCS*.

points_to_ocs(*points: Iterable[*Vec3*]*) → Iterable[*Vec3*]

Returns iterable of OCS vectors for UCS *points*.

The *OCS* is defined by the z-axis of the *UCS*.

Parameters

points – iterable of UCS vertices

to_ocs_angle_deg(*angle: float*) → float

Transforms *angle* from current UCS to the parent coordinate system (most likely the WCS) including the transformation to the OCS established by the extrusion vector *UCS.uz*.

Parameters

angle – in UCS in degrees

transform(*m:* Matrix44) → *UCS*

General inplace transformation interface, returns *self* (floating interface).

Parameters

m – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

rotate(*axis:* UVec, *angle: float*) → *UCS*

Returns a new rotated UCS, with the same origin as the source UCS. The rotation vector is located in the origin and has *WCS* coordinates e.g. (0, 0, 1) is the WCS z-axis as rotation vector.

Parameters

- **axis** – arbitrary rotation axis as vector in *WCS*
- **angle** – rotation angle in radians

rotate_local_x(*angle: float*) → *UCS*

Returns a new rotated UCS, rotation axis is the local x-axis.

Parameters

angle – rotation angle in radians

rotate_local_y(*angle: float*) → *UCS*

Returns a new rotated UCS, rotation axis is the local y-axis.

Parameters

angle – rotation angle in radians

rotate_local_z(*angle: float*) → *UCS*

Returns a new rotated UCS, rotation axis is the local z-axis.

Parameters

angle – rotation angle in radians

shift (*delta:* UVec) → *UCS*

Shifts current UCS by *delta* vector and returns *self*.

Parameters
delta – shifting vector

moveto (*location:* UVec) → *UCS*

Place current UCS at new origin *location* and returns *self*.

Parameters
location – new origin in WCS

static from_x_axis_and_point_in_xy (*origin:* UVec, *axis:* UVec, *point:* UVec) → *UCS*

Returns a new *UCS* defined by the origin, the x-axis vector and an arbitrary point in the xy-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – x-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in *WCS*

static from_x_axis_and_point_in_xz (*origin:* UVec, *axis:* UVec, *point:* UVec) → *UCS*

Returns a new *UCS* defined by the origin, the x-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – x-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in *WCS*

static from_y_axis_and_point_in_xy (*origin:* UVec, *axis:* UVec, *point:* UVec) → *UCS*

Returns a new *UCS* defined by the origin, the y-axis vector and an arbitrary point in the xy-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – y-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in *WCS*

static from_y_axis_and_point_in_yz (*origin:* UVec, *axis:* UVec, *point:* UVec) → *UCS*

Returns a new *UCS* defined by the origin, the y-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – y-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in *WCS*

static from_z_axis_and_point_in_xz (*origin:* UVec, *axis:* UVec, *point:* UVec) → *UCS*

Returns a new *UCS* defined by the origin, the z-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – z-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in *WCS*

static from_z_axis_and_point_in_yz(*origin:* UVec, *axis:* UVec, *point:* UVec) → *UCS*

Returns a new *UCS* defined by the origin, the z-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – z-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in *WCS*

render_axis(*layout:* BaseLayout, *length: float = 1*, *colors: RGB = (1, 3, 5)*)

Render axis as 3D lines into a *layout*.

Matrix44

class ezdxf.math.**Matrix44**(**args*)

An optimized 4x4 transformation matrix.

The utility functions for constructing transformations and transforming vectors and points assumes that vectors are stored as row vectors, meaning when multiplied, transformations are applied left to right (e.g. vAB transforms v by A then by B).

Matrix44 initialization:

- `Matrix44()` returns the identity matrix.
- `Matrix44(values)` values is an iterable with the 16 components of the matrix.
- `Matrix44(row1, row2, row3, row4)` four rows, each row with four values.

__repr__() → str

Returns the representation string of the matrix: `Matrix44((col0, col1, col2, col3), (...), (...), (...))`

get_row(*row: int*) → tuple[float, ...]

Get row as list of four float values.

Parameters

row – row index [0 .. 3]

set_row(*row: int*, *values: Sequence[float]*) → None

Sets the values in a row.

Parameters

- **row** – row index [0 .. 3]
- **values** – iterable of four row values

get_col(*col: int*) → tuple[float, ...]

Returns a column as a tuple of four floats.

Parameters

col – column index [0 .. 3]

set_col(*col: int*, *values: Sequence[float]*)

Sets the values in a column.

Parameters

- **col** – column index [0 .. 3]

- **values** – iterable of four column values

copy() → *Matrix44*

Returns a copy of same type.

__copy__() → *Matrix44*

Returns a copy of same type.

classmethod scale(*sx: float*, *sy: float | None = None*, *sz: float | None = None*) → *Matrix44*

Returns a scaling transformation matrix. If *sy* is `None`, *sy* = *sx*, and if *sz* is `None` *sz* = *sx*.

classmethod translate(*dx: float*, *dy: float*, *dz: float*) → *Matrix44*

Returns a translation matrix for translation vector (dx, dy, dz).

classmethod x_rotate(*angle: float*) → *Matrix44*

Returns a rotation matrix about the x-axis.

Parameters
angle – rotation angle in radians

classmethod y_rotate(*angle: float*) → *Matrix44*

Returns a rotation matrix about the y-axis.

Parameters
angle – rotation angle in radians

classmethod z_rotate(*angle: float*) → *Matrix44*

Returns a rotation matrix about the z-axis.

Parameters
angle – rotation angle in radians

classmethod axis_rotate(*axis:* UVec, *angle: float*) → *Matrix44*

Returns a rotation matrix about an arbitrary *axis*.

Parameters

- **axis** – rotation axis as `(x, y, z)` tuple or *Vec3* object
- **angle** – rotation angle in radians

classmethod xyz_rotate(*angle_x: float*, *angle_y: float*, *angle_z: float*) → *Matrix44*

Returns a rotation matrix for rotation about each axis.

Parameters

- **angle_x** – rotation angle about x-axis in radians
- **angle_y** – rotation angle about y-axis in radians
- **angle_z** – rotation angle about z-axis in radians

classmethod shear_xy(*angle_x: float = 0*, *angle_y: float = 0*) → *Matrix44*

Returns a translation matrix for shear mapping (visually similar to slanting) in the xy-plane.

Parameters

- **angle_x** – slanting angle in x direction in radians
- **angle_y** – slanting angle in y direction in radians

classmethod perspective_projection(*left: float*, *right: float*, *top: float*, *bottom: float*, *near: float*, *far: float*) → *Matrix44*

Returns a matrix for a 2D projection.

Parameters

- **left** – Coordinate of left of screen
- **right** – Coordinate of right of screen
- **top** – Coordinate of the top of the screen
- **bottom** – Coordinate of the bottom of the screen
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

classmethod perspective_projection_fov(*fov: float*, *aspect: float*, *near: float*, *far: float*) → *Matrix44*

Returns a matrix for a 2D projection.

Parameters

- **fov** – The field of view (in radians)
- **aspect** – The aspect ratio of the screen (width / height)
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

static chain(**matrices:* Matrix44) → *Matrix44*

Compose a transformation matrix from one or more *matrices*.

static ucs(*ux:* Vec3 = *X_AXIS*, *uy:* Vec3 = *Y_AXIS*, *uz:* Vec3 = *Z_AXIS*, *origin:* Vec3 = *NULLVEC*) → *Matrix44*

Returns a matrix for coordinate transformation from WCS to UCS. For transformation from UCS to WCS, transpose the returned matrix.

Parameters

- **ux** – x-axis for UCS as unit vector
- **uy** – y-axis for UCS as unit vector
- **uz** – z-axis for UCS as unit vector
- **origin** – UCS origin as location vector

__hash__()

Return hash(self).

__getitem__(*index: tuple[int, int]*)

Get (row, column) element.

__setitem__(*index: tuple[int, int]*, *value: float*)

Set (row, column) element.

__iter__() → Iterator[float]

Iterates over all matrix values.

rows() → Iterable[tuple[float, ...]]

Iterate over rows as 4-tuples.

columns() → Iterable[tuple[float, ...]]
Iterate over columns as 4-tuples.

__mul__(*other:* Matrix44) → *Matrix44*
Returns a new matrix as result of the matrix multiplication with another matrix.

__imul__(*other:* Matrix44) → *Matrix44*
Inplace multiplication with another matrix.

transform(*vector:* UVec) → *Vec3*
Returns a transformed vertex.

transform_direction(*vector:* UVec, *normalize=False*) → *Vec3*
Returns a transformed direction vector without translation.

transform_vertices(*vectors: Iterable[*UVec*]*) → Iterable[*Vec3*]
Returns an iterable of transformed vertices.

transform_directions(*vectors: Iterable[*UVec*]*, *normalize=False*) → Iterable[*Vec3*]
Returns an iterable of transformed direction vectors without translation.

transpose() → None
Swaps the rows for columns inplace.

determinant() → float
Returns determinant.

inverse() → None
Calculates the inverse of the matrix.

Raises
ZeroDivisionError – if matrix has no inverse.

property is_cartesian: bool
Returns `True` if target coordinate system is a right handed orthogonal coordinate system.

property is_orthogonal: bool
Returns `True` if target coordinate system has orthogonal axis.

Does not check for left- or right handed orientation, any orientation of the axis valid.

Basic Construction Classes

BoundingBox	3D bounding box.
BoundingBox2d	2D bounding box.
ConstructionArc	Construction tool for 2D arcs.
ConstructionBox	Construction tool for 2D rectangles.
ConstructionCircle	Construction tool for 2D circles.
ConstructionEllipse	Construction tool for 3D ellipsis.
ConstructionLine	Construction tool for 2D lines.
ConstructionPolyline	Construction tool for 3D polylines.
ConstructionRay	Construction tool for infinite 2D rays.
Plane	Construction tool for 3D planes.
Shape2d	Construction tools for 2D shapes.
Vec2	Immutable 2D vector class.
Vec3	Immutable 3D vector class.

UVec

class ezdxf.math.**UVec**

Type alias for `Union[Sequence[float], Vec2, Vec3]`

Vec3

class ezdxf.math.**Vec3**(**args*)

Immutable 3D vector class.

This class is optimized for universality not for speed. Immutable means you can't change (x, y, z) components after initialization:

```
v1 = Vec3(1, 2, 3)
v2 = v1
v2.z = 7  # this is not possible, raises AttributeError
v2 = Vec3(v2.x, v2.y, 7)  # this creates a new Vec3() object
assert v1.z == 3  # and v1 remains unchanged
```

Vec3 initialization:

- `Vec3()`, returns `Vec3(0, 0, 0)`
- `Vec3((x, y))`, returns `Vec3(x, y, 0)`
- `Vec3((x, y, z))`, returns `Vec3(x, y, z)`
- `Vec3(x, y)`, returns `Vec3(x, y, 0)`
- `Vec3(x, y, z)`, returns `Vec3(x, y, z)`

Addition, subtraction, scalar multiplication and scalar division left and right-handed are supported:

```
v = Vec3(1, 2, 3)
v + (1, 2, 3) == Vec3(2, 4, 6)
(1, 2, 3) + v == Vec3(2, 4, 6)
v - (1, 2, 3) == Vec3(0, 0, 0)
(1, 2, 3) - v == Vec3(0, 0, 0)
v * 3 == Vec3(3, 6, 9)
3 * v == Vec3(3, 6, 9)
Vec3(3, 6, 9) / 3 == Vec3(1, 2, 3)
-Vec3(1, 2, 3) == (-1, -2, -3)
```

Comparison between vectors and vectors or tuples is supported:

```
Vec3(1, 2, 3) < Vec3 (2, 2, 2)
(1, 2, 3) < tuple(Vec3(2, 2, 2))  # conversion necessary
Vec3(1, 2, 3) == (1, 2, 3)

bool(Vec3(1, 2, 3)) is True
bool(Vec3(0, 0, 0)) is False
```

x

x-axis value

y

y-axis value

z

z-axis value

xy

Vec3 as `(x, y, 0)`, projected on the xy-plane.

xyz

Vec3 as `(x, y, z)` tuple.

vec2

Real 2D vector as *Vec2* object.

magnitude

Length of vector.

magnitude_xy

Length of vector in the xy-plane.

magnitude_square

Square length of vector.

is_null

`Vec3(0, 0, 0)`. Has a fixed absolute testing tolerance of 1e-12!

Type
`True` if all components are close to zero

angle

Angle between vector and x-axis in the xy-plane in radians.

angle_deg

Returns angle of vector and x-axis in the xy-plane in degrees.

spatial_angle

Spatial angle between vector and x-axis in radians.

spatial_angle_deg

Spatial angle between vector and x-axis in degrees.

__str__() → str

Return `'(x, y, z)'` as string.

__repr__() → str

Return `'Vec3(x, y, z)'` as string.

__len__() → int

Returns always `3`.

__hash__() → int

Returns hash value of vector, enables the usage of vector as key in `set` and `dict`.

copy() → *Vec3*

Returns a copy of vector as *Vec3* object.

__copy__() → *Vec3*

Returns a copy of vector as *Vec3* object.

__deepcopy__(*memodict: dict*) → *Vec3*

`copy.deepcopy()` support.

__getitem__(*index: int*) → float

Support for indexing:

- v[0] is v.x
- v[1] is v.y
- v[2] is v.z

__iter__() → Iterator[float]

Returns iterable of x-, y- and z-axis.

__abs__() → float

Returns length (magnitude) of vector.

replace(*x: float | None = None*, *y: float | None = None*, *z: float | None = None*) → *Vec3*

Returns a copy of vector with replaced x-, y- and/or z-axis.

classmethod generate(*items: Iterable[*UVec*]*) → Iterable[*Vec3*]

Returns an iterable of `Vec3` objects.

classmethod list(*items: Iterable[*UVec*]*) → list[*ezdxf.math._vector.Vec3*]

Returns a list of `Vec3` objects.

classmethod tuple(*items: Iterable[*UVec*]*) → Sequence[*Vec3*]

Returns a tuple of `Vec3` objects.

classmethod from_angle(*angle: float*, *length: float = 1.0*) → *Vec3*

Returns a `Vec3` object from *angle* in radians in the xy-plane, z-axis = `0`.

classmethod from_deg_angle(*angle: float*, *length: float = 1.0*) → *Vec3*

Returns a `Vec3` object from *angle* in degrees in the xy-plane, z-axis = `0`.

orthogonal(*ccw: bool = True*) → *Vec3*

Returns orthogonal 2D vector, z-axis is unchanged.

Parameters

ccw – counter-clockwise if `True` else clockwise

lerp(*other:* UVec, *factor=0.5*) → *Vec3*

Returns linear interpolation between *self* and *other*.

Parameters

- **other** – end point as `Vec3` compatible object
- **factor** – interpolation factor (0 = self, 1 = other, 0.5 = mid point)

is_parallel(*other:* Vec3, *, *rel_tol: float = 1e-9*, *abs_tol: float = 1e-12*) → bool

Returns `True` if *self* and *other* are parallel to vectors.

project(*other:* UVec) → *Vec3*

Returns projected vector of *other* onto *self*.

normalize(*length: float = 1.0*) → *Vec3*

Returns normalized vector, optional scaled by *length*.

reversed() → *Vec3*

Returns negated vector (*-self*).

isclose(*other:* UVec, *, *rel_tol: float = 1e-9*, *abs_tol: float = 1e-12*) → bool

Returns `True` if *self* is close to *other*. Uses `math.isclose()` to compare all axis.

Learn more about the `math.isclose()` function in PEP 485.

__neg__() → *Vec3*

Returns negated vector (*-self*).

__bool__() → bool

Returns `True` if vector is not (0, 0, 0).

__eq__(*other:* UVec) → bool

Equal operator.

Parameters
other – *Vec3* compatible object

__lt__(*other:* UVec) → bool

Lower than operator.

Parameters
other – *Vec3* compatible object

__add__(*other:* UVec) → *Vec3*

Add *Vec3* operator: *self* + *other*.

__radd__(*other:* UVec) → *Vec3*

RAdd *Vec3* operator: *other* + *self*.

__sub__(*other:* UVec) → *Vec3*

Sub *Vec3* operator: *self* - *other*.

__rsub__(*other:* UVec) → *Vec3*

RSub *Vec3* operator: *other* - *self*.

__mul__(*other: float*) → *Vec3*

Scalar Mul operator: *self* * *other*.

__rmul__(*other: float*) → *Vec3*

Scalar RMul operator: *other* * *self*.

__truediv__(*other: float*) → *Vec3*

Scalar Div operator: *self* / *other*.

dot(*other:* UVec) → float

Dot operator: *self* . *other*

Parameters
other – *Vec3* compatible object

cross(*other:* UVec) → *Vec3*

Cross operator: *self* x *other*

Parameters
other – *Vec3* compatible object

distance(*other:* UVec) → float

Returns distance between *self* and *other* vector.

angle_about(*base:* UVec, *target:* UVec) → float

Returns counter-clockwise angle in radians about *self* from *base* to *target* when projected onto the plane defined by *self* as the normal vector.

Parameters

- **base** – base vector, defines angle 0
- **target** – target vector

angle_between(*other:* UVec) → float

Returns angle between *self* and *other* in radians. +angle is counter clockwise orientation.

Parameters
other – *Vec3* compatible object

rotate(*angle: float*) → *Vec3*

Returns vector rotated about *angle* around the z-axis.

Parameters
angle – angle in radians

rotate_deg(*angle: float*) → *Vec3*

Returns vector rotated about *angle* around the z-axis.

Parameters
angle – angle in degrees

static sum(*items: Iterable[*UVec*]*) → *Vec3*

Add all vectors in *items*.

ezdxf.math.**X_AXIS**

```
Vec3(1, 0, 0)
```

ezdxf.math.**Y_AXIS**

```
Vec3(0, 1, 0)
```

ezdxf.math.**Z_AXIS**

```
Vec3(0, 0, 1)
```

ezdxf.math.**NULLVEC**

```
Vec3(0, 0, 0)
```

Vec2

class ezdxf.math.**Vec2**(*v=(0.0, 0.0)*, *y=None*)

Immutable 2D vector class.

Parameters

- **v** – vector object with `x` and `y` attributes/properties or a sequence of float `[x, y, ...]` or x-axis as float if argument *y* is not `None`
- **y** – second float for `Vec2(x, y)`

Vec2 implements a subset of *Vec3*.

Plane

class ezdxf.math.**Plane**(*normal:* Vec3, *distance: float*)

Construction tool for 3D planes.

Represents a plane in 3D space as a normal vector and the perpendicular distance from the origin.

normal

Normal vector of the plane.

distance_from_origin

The (perpendicular) distance of the plane from origin (0, 0, 0).

vector

Returns the location vector.

classmethod from_3p(*a:* Vec3, *b:* Vec3, *c:* Vec3) → *Plane*

Returns a new plane from 3 points in space.

classmethod from_vector(*vector:* UVec) → *Plane*

Returns a new plane from the given location vector.

copy() → *Plane*

Returns a copy of the plane.

signed_distance_to(*v:* Vec3) → float

Returns signed distance of vertex *v* to plane, if distance is > 0, *v* is in 'front' of plane, in direction of the normal vector, if distance is < 0, *v* is at the 'back' of the plane, in the opposite direction of the normal vector.

distance_to(*v:* Vec3) → float

Returns absolute (unsigned) distance of vertex *v* to plane.

is_coplanar_vertex(*v:* Vec3, *abs_tol=1e-9*) → bool

Returns `True` if vertex *v* is coplanar, distance from plane to vertex *v* is 0.

is_coplanar_plane(*p:* Plane, *abs_tol=1e-9*) → bool

Returns `True` if plane *p* is coplanar, normal vectors in same or opposite direction.

intersect_line(*start:* Vec3, *end:* Vec3, *, *coplanar=True*, *abs_tol=PLANE_EPSILON*) → *Vec3* | None

Returns the intersection point of the 3D line from *start* to *end* and this plane or `None` if there is no intersection. If the argument *coplanar* is `False` the start- or end point of the line are ignored as intersection points.

intersect_ray(*origin:* Vec3, *direction:* Vec3) → *Vec3* | None

Returns the intersection point of the infinite 3D ray defined by *origin* and the *direction* vector and this plane or `None` if there is no intersection. A coplanar ray does not intersect the plane!

BoundingBox

class ezdxf.math.**BoundingBox**(*vertices: Iterable[*UVec*]* | *None = None*)

3D bounding box.

Parameters
vertices – iterable of `(x, y, z)` tuples or *Vec3* objects

extmin

"lower left" corner of bounding box

extmax

"upper right" corner of bounding box

property is_empty: bool

Returns `True` if the bounding box is empty or the bounding box has a size of 0 in any or all dimensions or is undefined.

property has_data: bool

Returns `True` if the bonding box has known limits.

property size

Returns size of bounding box.

property center

Returns center of bounding box.

inside(*vertex:* UVec) → bool

Returns `True` if *vertex* is inside this bounding box.

Vertices at the box border are inside!

any_inside(*vertices: Iterable[*UVec*]*) → bool

Returns `True` if any vertex is inside this bounding box.

Vertices at the box border are inside!

all_inside(*vertices: Iterable[*UVec*]*) → bool

Returns `True` if all vertices are inside this bounding box.

Vertices at the box border are inside!

has_intersection(*other: AbstractBoundingBox*) → bool

Returns `True` if this bounding box intersects with *other* but does not include touching bounding boxes, see also *has_overlap()*:

```
bbox1 = BoundingBox([(0, 0, 0), (1, 1, 1)])
bbox2 = BoundingBox([(1, 1, 1), (2, 2, 2)])
assert bbox1.has_intersection(bbox2) is False
```

has_overlap(*other: AbstractBoundingBox*) → bool

Returns `True` if this bounding box intersects with *other* but in contrast to *has_intersection()* includes touching bounding boxes too:

```
bbox1 = BoundingBox([(0, 0, 0), (1, 1, 1)])
bbox2 = BoundingBox([(1, 1, 1), (2, 2, 2)])
assert bbox1.has_overlap(bbox2) is True
```

contains(*other: AbstractBoundingBox*) → bool

Returns `True` if the *other* bounding box is completely inside this bounding box.

extend(*vertices: Iterable[*UVec*]*) → None

Extend bounds by *vertices*.

Parameters
vertices – iterable of vertices

union(*other: AbstractBoundingBox*)

Returns a new bounding box as union of this and *other* bounding box.

intersection(*other: AbstractBoundingBox*) → *BoundingBox*

Returns the bounding box of the intersection cube of both 3D bounding boxes. Returns an empty bounding box if the intersection volume is 0.

rect_vertices() → Sequence[*Vec2*]

Returns the corners of the bounding box in the xy-plane as `Vec2` objects.

cube_vertices() → Sequence[*Vec3*]

Returns the 3D corners of the bounding box as `Vec3` objects.

grow(*value: float*) → None

Grow or shrink the bounding box by an uniform value in x, y and z-axis. A negative value shrinks the bounding box. Raises `ValueError` for shrinking the size of the bounding box to zero or below in any dimension.

BoundingBox2d

class ezdxf.math.**BoundingBox2d**(*vertices: Iterable[*UVec*] | None = None*)

2D bounding box.

Parameters
vertices – iterable of `(x, y[, z])` tuples or `Vec3` objects

extmin

"lower left" corner of bounding box

extmax

"upper right" corner of bounding box

property is_empty: bool

Returns `True` if the bounding box is empty. The bounding box has a size of 0 in any or all dimensions or is undefined.

property has_data: bool

Returns `True` if the bonding box has known limits.

property size

Returns size of bounding box.

property center

Returns center of bounding box.

inside(*vertex:* UVec) → bool

Returns `True` if *vertex* is inside this bounding box.

Vertices at the box border are inside!

any_inside(*vertices: Iterable[*UVec*]*) → bool

Returns `True` if any vertex is inside this bounding box.

Vertices at the box border are inside!

all_inside(*vertices: Iterable[*UVec*]*) → bool

Returns `True` if all vertices are inside this bounding box.

Vertices at the box border are inside!

has_intersection(*other: AbstractBoundingBox*) → bool

Returns `True` if this bounding box intersects with *other* but does not include touching bounding boxes, see also *has_overlap()*:

```
bbox1 = BoundingBox2d([(0, 0), (1, 1)])
bbox2 = BoundingBox2d([(1, 1), (2, 2)])
assert bbox1.has_intersection(bbox2) is False
```

has_overlap(*other: AbstractBoundingBox*) → bool

Returns `True` if this bounding box intersects with *other* but in contrast to *has_intersection()* includes touching bounding boxes too:

```
bbox1 = BoundingBox2d([(0, 0), (1, 1)])
bbox2 = BoundingBox2d([(1, 1), (2, 2)])
assert bbox1.has_overlap(bbox2) is True
```

contains(*other: AbstractBoundingBox*) → bool

Returns `True` if the *other* bounding box is completely inside this bounding box.

extend(*vertices: Iterable[*UVec*]*) → None

Extend bounds by *vertices*.

Parameters
vertices – iterable of vertices

union(*other: AbstractBoundingBox*)

Returns a new bounding box as union of this and *other* bounding box.

intersection(*other: AbstractBoundingBox*) → *BoundingBox2d*

Returns the bounding box of the intersection rectangle of both 2D bounding boxes. Returns an empty bounding box if the intersection area is 0.

rect_vertices() → Sequence[*Vec2*]

Returns the corners of the bounding box in the xy-plane as *Vec2* objects.

ConstructionRay

class ezdxf.math.**ConstructionRay**(*p1:* UVec, *p2:* UVec | *None* = *None*, *angle: float* | *None* = *None*)

Construction tool for infinite 2D rays.

Parameters

- **p1** – definition point 1
- **p2** – ray direction as 2nd point or `None`
- **angle** – ray direction as angle in radians or `None`

location

Location vector as *Vec2*.

direction

Direction vector as *Vec2*.

slope

Slope of ray or `None` if vertical.

angle
Angle between x-axis and ray in radians.

angle_deg
Angle between x-axis and ray in degrees.

is_vertical
`True` if ray is vertical (parallel to y-axis).

is_horizontal
`True` if ray is horizontal (parallel to x-axis).

__str__()
Return str(self).

is_parallel(*other:* ConstructionRay) → bool
Returns `True` if rays are parallel.

intersect(*other:* ConstructionRay) → *Vec2*
Returns the intersection point as `(x, y)` tuple of *self* and *other*.

Raises
ParallelRaysError – if rays are parallel

orthogonal(*location:* UVec) → *ConstructionRay*
Returns orthogonal ray at *location*.

bisectrix(*other:* ConstructionRay) → *ConstructionRay*
Bisectrix between *self* and *other*.

yof(*x: float*) → float
Returns y-value of ray for *x* location.

Raises
ArithmeticError – for vertical rays

xof(*y: float*) → float
Returns x-value of ray for *y* location.

Raises
ArithmeticError – for horizontal rays

ConstructionLine

class ezdxf.math.**ConstructionLine**(*start:* UVec, *end:* UVec)
Construction tool for 2D lines.

The *ConstructionLine* class is similar to *ConstructionRay*, but has a start- and endpoint. The direction of line goes from start- to endpoint, "left of line" is always in relation to this line direction.

Parameters

- **start** – start point of line as *Vec2* compatible object
- **end** – end point of line as *Vec2* compatible object

start
start point as *Vec2*

end
end point as *Vec2*

bounding_box
bounding box of line as *BoundingBox2d* object.

ray
collinear *ConstructionRay*.

is_vertical
`True` if line is vertical.

is_horizontal
`True` if line is horizontal.

__str__()
Return str(self).

translate(*dx: float*, *dy: float*) → None
Move line about *dx* in x-axis and about *dy* in y-axis.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

length() → float
Returns length of line.

midpoint() → *Vec2*
Returns mid point of line.

inside_bounding_box(*point:* UVec) → bool
Returns `True` if *point* is inside of line bounding box.

intersect(*other:* ConstructionLine, *abs_tol: float = TOLERANCE*) → *Vec2* | None
Returns the intersection point of to lines or `None` if they have no intersection point.

Parameters

- **other** – other *ConstructionLine*
- **abs_tol** – tolerance for distance check

has_intersection(*other:* ConstructionLine, *abs_tol: float = TOLERANCE*) → bool
Returns `True` if has intersection with *other* line.

is_point_left_of_line(*point:* UVec, *colinear=False*) → bool
Returns `True` if *point* is left of construction line in relation to the line direction from start to end.

If *colinear* is `True`, a colinear point is also left of the line.

ConstructionCircle

class ezdxf.math.**ConstructionCircle**(*center:* UVec, *radius: float = 1.0*)

Construction tool for 2D circles.

Parameters

- **center** – center point as *Vec2* compatible object
- **radius** – circle radius > *0*

center

center point as *Vec2*

radius

radius as float

bounding_box

2D bounding box of circle as *BoundingBox2d* object.

static from_3p(*p1:* UVec, *p2:* UVec, *p3:* UVec) → *ConstructionCircle*

Creates a circle from three points, all points have to be compatible to *Vec2* class.

__str__() → str

Returns string representation of circle "ConstructionCircle(center, radius)".

translate(*dx: float*, *dy: float*) → None

Move circle about *dx* in x-axis and about *dy* in y-axis.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

point_at(*angle: float*) → *Vec2*

Returns point on circle at *angle* as *Vec2* object.

Parameters

angle – angle in radians, angle goes counter clockwise around the z-axis, x-axis = 0 deg.

vertices(*angles: Iterable[float]*) → Iterable[*Vec2*]

Yields vertices of the circle for iterable *angles*.

Parameters

angles – iterable of angles as radians, angle goes counter-clockwise around the z-axis, x-axis = 0 deg.

flattening(*sagitta: float*) → Iterator[*Vec2*]

Approximate the circle by vertices, argument *sagitta* is the max. distance from the center of an arc segment to the center of its chord. Returns a closed polygon where the start vertex is coincident with the end vertex!

inside(*point:* UVec) → bool

Returns `True` if *point* is inside circle.

tangent(*angle: float*) → *ConstructionRay*

Returns tangent to circle at *angle* as *ConstructionRay* object.

Parameters

angle – angle in radians

intersect_ray(*ray:* ConstructionRay, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of circle and *ray* as sequence of *Vec2* objects.

Parameters

- **ray** – intersection ray
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	ray is a tangent to circle
2	ray intersects with the circle

intersect_line(*line:* ConstructionLine, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of circle and *line* as sequence of *Vec2* objects.

Parameters

- **line** – intersection line
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	line intersects or touches the circle at one point
2	line intersects the circle at two points

intersect_circle(*other:* ConstructionCircle, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of two circles as sequence of *Vec2* objects.

Parameters

- **other** – intersection circle
- **abs_tol** – absolute tolerance for tests

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	circle touches the *other* circle at one point
2	circle intersects with the *other* circle

ConstructionArc

class ezdxf.math.**ConstructionArc**(*center: UVec = (0, 0)*, *radius: float = 1.0*, *start_angle: float = 0.0*, *end_angle: float = 360.0*, *is_counter_clockwise: bool | None = True*)

Construction tool for 2D arcs.

ConstructionArc represents a 2D arc in the xy-plane, use an *UCS* to place a DXF *Arc* entity in 3D space, see method *add_to_layout()*.

Implements the 2D transformation tools: *translate()*, *scale_uniform()* and *rotate_z()*

Parameters

- **center** – center point as *Vec2* compatible object
- **radius** – radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **is_counter_clockwise** – swaps start- and end angle if `False`

center

center point as *Vec2*

radius

radius as float

start_angle

start angle in degrees

end_angle

end angle in degrees

angle_span

Returns angle span of arc from start- to end param.

start_angle_rad

Returns the start angle in radians.

end_angle_rad

Returns the end angle in radians.

start_point

start point of arc as *Vec2*.

end_point

end point of arc as *Vec2*.

bounding_box

bounding box of arc as *BoundingBox2d*.

angles(*num: int*) → Iterable[float]

Returns *num* angles from start- to end angle in degrees in counter-clockwise order.

All angles are normalized in the range from [0, 360).

vertices (*a: Iterable[float]*) → Iterable[*Vec2*]

Yields vertices on arc for angles in iterable *a* in WCS as location vectors.

Parameters
a – angles in the range from 0 to 360 in degrees, arc goes counter clockwise around the z-axis, WCS x-axis = 0 deg.

tangents (*a: Iterable[float]*) → Iterable[*Vec2*]

Yields tangents on arc for angles in iterable *a* in WCS as direction vectors.

Parameters
a – angles in the range from 0 to 360 in degrees, arc goes counter-clockwise around the z-axis, WCS x-axis = 0 deg.

translate (*dx: float*, *dy: float*) → *ConstructionArc*

Move arc about *dx* in x-axis and about *dy* in y-axis, returns *self* (floating interface).

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

scale_uniform (*s: float*) → *ConstructionArc*

Scale arc inplace uniform about *s* in x- and y-axis, returns *self* (floating interface).

rotate_z (*angle: float*) → *ConstructionArc*

Rotate arc inplace about z-axis, returns *self* (floating interface).

Parameters
angle – rotation angle in degrees

classmethod from_2p_angle (*start_point:* UVec, *end_point:* UVec, *angle: float*, *ccw: bool = True*) → *ConstructionArc*

Create arc from two points and enclosing angle. Additional precondition: arc goes by default in counter-clockwise orientation from *start_point* to *end_point*, can be changed by *ccw* = `False`.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **angle** – enclosing angle in degrees
- **ccw** – counter-clockwise direction if `True`

classmethod from_2p_radius (*start_point:* UVec, *end_point:* UVec, *radius: float*, *ccw: bool = True*, *center_is_left: bool = True*) → *ConstructionArc*

Create arc from two points and arc radius. Additional precondition: arc goes by default in counter-clockwise orientation from *start_point* to *end_point* can be changed by *ccw* = `False`.

The parameter *center_is_left* defines if the center of the arc is left or right of the line from *start_point* to *end_point*. Parameter *ccw* = `False` swaps start- and end point, which also inverts the meaning of `center_is_left`.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **radius** – arc radius

- **ccw** – counter-clockwise direction if `True`
- **center_is_left** – center point of arc is left of line from start- to end point if `True`

classmethod from_3p(*start_point:* UVec, *end_point:* UVec, *def_point:* UVec, *ccw: bool = True*) → *ConstructionArc*

Create arc from three points. Additional precondition: arc goes in counter-clockwise orientation from *start_point* to *end_point*.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **def_point** – additional definition point as *Vec2* compatible object
- **ccw** – counter-clockwise direction if `True`

add_to_layout(*layout:* BaseLayout, *ucs:* UCS | *None = None*, *dxfattribs=None*) → *Arc*

Add arc as DXF *Arc* entity to a layout.

Supports 3D arcs by using an *UCS*. An *ConstructionArc* is always defined in the xy-plane, but by using an arbitrary UCS, the arc can be placed in 3D space, automatically OCS transformation included.

Parameters

- **layout** – destination layout as *BaseLayout* object
- **ucs** – place arc in 3D space by *UCS* object
- **dxfattribs** – additional DXF attributes for the ARC entity

intersect_ray(*ray:* ConstructionRay, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of arc and *ray* as sequence of *Vec2* objects.

Parameters

- **ray** – intersection ray
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	line intersects or touches the arc at one point
2	line intersects the arc at two points

intersect_line(*line:* ConstructionLine, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of arc and *line* as sequence of *Vec2* objects.

Parameters

- **line** – intersection line
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	line intersects or touches the arc at one point
2	line intersects the arc at two points

intersect_circle(*circle:* ConstructionCircle, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of arc and *circle* as sequence of *Vec2* objects.

Parameters

- **circle** – intersection circle
- **abs_tol** – absolute tolerance for tests

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	circle intersects or touches the arc at one point
2	circle intersects the arc at two points

intersect_arc(*other:* ConstructionArc, *abs_tol: float = 1e-10*) → Sequence[*Vec2*]

Returns intersection points of two arcs as sequence of *Vec2* objects.

Parameters

- **other** – other intersection arc
- **abs_tol** – absolute tolerance for tests

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	other arc intersects or touches the arc at one point
2	other arc intersects the arc at two points

ConstructionEllipse

class ezdxf.math.**ConstructionEllipse**(*center:* UVec = *NULLVEC*, *major_axis:* UVec = *X_AXIS*, *extrusion:* UVec = *Z_AXIS*, *ratio: float = 1*, *start_param: float = 0*, *end_param: float = math.tau*, *ccw: bool = True*)

Construction tool for 3D ellipsis.

Parameters

- **center** – 3D center point
- **major_axis** – major axis as 3D vector

- **extrusion** – normal vector of ellipse plane
- **ratio** – ratio of minor axis to major axis
- **start_param** – start param in radians
- **end_param** – end param in radians
- **ccw** – is counter-clockwise flag - swaps start- and end param if `False`

center

center point as *Vec3*

major_axis

major axis as *Vec3*

minor_axis

minor axis as *Vec3*, automatically calculated from *major_axis* and *extrusion*.

extrusion

extrusion vector (normal of ellipse plane) as *Vec3*

ratio

ratio of minor axis to major axis (float)

start

start param in radians (float)

end

end param in radians (float)

start_point

Returns start point of ellipse as Vec3.

end_point

Returns end point of ellipse as Vec3.

property param_span: float

Returns the counter-clockwise params span from start- to end param, see also *ezdxf.math.ellipse_param_span()* for more information.

to_ocs() → *ConstructionEllipse*

Returns ellipse parameters as OCS representation.

OCS elevation is stored in `center.z`.

params(*num: int*) → Iterable[float]

Returns *num* params from start- to end param in counter-clockwise order.

All params are normalized in the range from [0, 2π).

vertices(*params: Iterable[float]*) → Iterable[*Vec3*]

Yields vertices on ellipse for iterable *params* in WCS.

Parameters

params – param values in the range from [0, 2π) in radians, param goes counter-clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

flattening(*distance: float*, *segments: int = 4*) → Iterable[*Vec3*]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided. Returns a closed polygon for a full ellipse: start vertex == end vertex.

Parameters

- **distance** – maximum distance from the projected curve point onto the segment chord.
- **segments** – minimum segment count

params_from_vertices(*vertices: Iterable[UVec]*) → Iterable[float]

Yields ellipse params for all given *vertices*.

The vertex don't have to be exact on the ellipse curve or in the range from start- to end param or even in the ellipse plane. Param is calculated from the intersection point of the ray projected on the ellipse plane from the center of the ellipse through the vertex.

Warning: An input for start- and end vertex at param 0 and 2π return unpredictable results because of floating point inaccuracy, sometimes 0 and sometimes 2π.

dxfattribs() → dict[str, Any]

Returns required DXF attributes to build an ELLIPSE entity.

Entity ELLIPSE has always a ratio in range from 1e-6 to 1.

main_axis_points() → Iterable[*Vec3*]

Yields main axis points of ellipse in the range from start- to end param.

classmethod from_arc(*center: UVec = NULLVEC*, *radius: float = 1*, *extrusion: UVec = Z_AXIS*, *start_angle: float = 0*, *end_angle: float = 360*, *ccw: bool = True*) → *ConstructionEllipse*

Returns `ConstructionEllipse` from arc or circle.

Arc and Circle parameters defined in OCS.

Parameters

- **center** – center in OCS
- **radius** – arc or circle radius
- **extrusion** – OCS extrusion vector
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **ccw** – arc curve goes counter clockwise from start to end if `True`

transform(*m: Matrix44*) → None

Transform ellipse in place by transformation matrix *m*.

swap_axis() → None

Swap axis and adjust start- and end parameter.

add_to_layout(*layout: BaseLayout*, *dxfattribs=None*) → *Ellipse*

Add ellipse as DXF `Ellipse` entity to a layout.

Parameters

- **layout** – destination layout as *BaseLayout* object
- **dxfattribs** – additional DXF attributes for the ELLIPSE entity

ConstructionBox

class ezdxf.math.**ConstructionBox**(*center:* UVec = *(0, 0)*, *width: float = 1*, *height: float = 1*, *angle: float = 0*)

Construction tool for 2D rectangles.

Parameters

- **center** – center of rectangle
- **width** – width of rectangle
- **height** – height of rectangle
- **angle** – angle of rectangle in degrees

center

box center

width

box width

height

box height

angle

rotation angle in degrees

corners

box corners as sequence of *Vec2* objects.

bounding_box

BoundingBox2d

incircle_radius

incircle radius

circumcircle_radius

circum circle radius

__iter__() → Iterable[*Vec2*]

Iterable of box corners as *Vec2* objects.

__getitem__(*corner*) → *Vec2*

Get corner by index *corner*, `list` like slicing is supported.

__repr__() → str

Returns string representation of box as `ConstructionBox(center, width, height, angle)`

classmethod from_points(*p1:* UVec, *p2:* UVec) → *ConstructionBox*

Creates a box from two opposite corners, box sides are parallel to x- and y-axis.

Parameters

- **p1** – first corner as *Vec2* compatible object

- **p2** – second corner as *Vec2* compatible object

translate(*dx: float*, *dy: float*) → None

Move box about *dx* in x-axis and about *dy* in y-axis.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

expand(*dw: float*, *dh: float*) → None

Expand box: *dw* expand width, *dh* expand height.

scale(*sw: float*, *sh: float*) → None

Scale box: *sw* scales width, *sh* scales height.

rotate(*angle: float*) → None

Rotate box by *angle* in degrees.

is_inside(*point:* UVec) → bool

Returns `True` if *point* is inside of box.

is_any_corner_inside(*other:* ConstructionBox) → bool

Returns `True` if any corner of *other* box is inside this box.

is_overlapping(*other:* ConstructionBox) → bool

Returns `True` if this box and *other* box do overlap.

border_lines() → Sequence[*ConstructionLine*]

Returns borderlines of box as sequence of *ConstructionLine*.

intersect(*line:* ConstructionLine) → list[*ezdxf.math._vector.Vec2*]

Returns 0, 1 or 2 intersection points between *line* and box borderlines.

Parameters

line – line to intersect with borderlines

Returns

list of intersection points

list size	Description
0	no intersection
1	line touches box at one corner
2	line intersects with box

ConstructionPolyline

class ezdxf.math.**ConstructionPolyline**(*vertices: Iterable[*UVec*]*, *close: bool = False*, *rel_tol: float = REL_TOL*)

Construction tool for 3D polylines.

A polyline construction tool to measure, interpolate and divide anything that can be approximated or flattened into vertices. This is an immutable data structure which supports the `Sequence` interface.

Parameters

- **vertices** – iterable of polyline vertices
- **close** – `True` to close the polyline (first vertex == last vertex)
- **rel_tol** – relative tolerance for floating point comparisons

Example to measure or divide a SPLINE entity:

```
import ezdxf
from ezdxf.math import ConstructionPolyline

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
spline = msp.query("SPLINE").first
if spline is not None:
    polyline = ConstructionPolyline(spline.flattening(0.01))
    print(f"Entity {spline} has an approximated length of {polyline.length}")
    # get dividing points with a distance of 1.0 drawing unit to each other
    points = list(polyline.divide_by_length(1.0))
```

property length: float

Returns the overall length of the polyline.

property is_closed: bool

Returns `True` if the polyline is closed (first vertex == last vertex).

data(*index: int*) → tuple[float, float, *ezdxf.math._vector.Vec3*]

Returns the tuple (distance from start, distance from previous vertex, vertex). All distances measured along the polyline.

index_at(*distance: float*) → int

Returns the data index of the exact or next data entry for the given *distance*. Returns the index of last entry if *distance* > `length`.

vertex_at(*distance: float*) → *Vec3*

Returns the interpolated vertex at the given *distance* from the start of the polyline.

divide(*count: int*) → Iterator[*Vec3*]

Returns *count* interpolated vertices along the polyline. Argument *count* has to be greater than 2 and the start- and end vertices are always included.

divide_by_length(*length: float*, *force_last: bool = False*) → Iterator[*Vec3*]

Returns interpolated vertices along the polyline. Each vertex has a fix distance *length* from its predecessor. Yields the last vertex if argument *force_last* is `True` even if the last distance is not equal to *length*.

Shape2d

class ezdxf.math.**Shape2d**(*vertices: Iterable[UVec] | None = None*)

Construction tools for 2D shapes.

A 2D geometry object as list of `Vec2` objects, vertices can be moved, rotated and scaled.

Parameters
vertices – iterable of `Vec2` compatible objects.

vertices

List of `Vec2` objects

bounding_box

BoundingBox2d

__len__() → int

Returns *count* of vertices.

__getitem__(*item: int | slice*) → *Vec2*

Get vertex by index *item*, supports `list` like slicing.

append(*vertex:* UVec) → None

Append single *vertex*.

Parameters

vertex – vertex as *Vec2* compatible object

extend(*vertices: Iterable*) → None

Append multiple *vertices*.

Parameters

vertices – iterable of vertices as *Vec2* compatible objects

translate(*vector:* UVec) → None

Translate shape about *vector*.

scale(*sx: float = 1.0*, *sy: float = 1.0*) → None

Scale shape about *sx* in x-axis and *sy* in y-axis.

scale_uniform(*scale: float*) → None

Scale shape uniform about *scale* in x- and y-axis.

rotate(*angle: float*, *center:* UVec | *None = None*) → None

Rotate shape around rotation *center* about *angle* in degrees.

rotate_rad(*angle: float*, *center:* UVec | *None = None*) → None

Rotate shape around rotation *center* about *angle* in radians.

offset(*offset: float*, *closed: bool = False*) → *Shape2d*

Returns a new offset shape, for more information see also *ezdxf.math.offset_vertices_2d()* function.

Parameters

- **offset** – line offset perpendicular to direction of shape segments defined by vertices order, offset > `0` is 'left' of line segment, offset < `0` is 'right' of line segment
- **closed** – `True` to handle as closed shape

convex_hull() → *Shape2d*

Returns convex hull as new shape.

Curves

ApproxParamT	Approximation tool for parametrized curves.
BSpline	B-spline construction tool.
Bezier	Generic Bézier curve of any degree.
Bezier3P	Implements an optimized quadratic Bézier curve for exact 3 control points.
Bezier4P	Implements an optimized cubic Bézier curve for exact 4 control points.
BezierSurface	*BezierSurface* defines a mesh of *m* x *n* control points.
EulerSpiral	This class represents an euler spiral (clothoid) for *curvature* (Radius of curvature).

BSpline

class ezdxf.math.**BSpline**(*control_points: Iterable[UVec], order: int = 4, knots: Iterable[float] | None = None, weights: Iterable[float] | None = None*)

B-spline construction tool.

Internal representation of a B-spline curve. The default configuration of the knot vector is a uniform open knot vector ("clamped").

Factory functions:

- *fit_points_to_cad_cv()*
- *fit_points_to_cubic_bezier()*
- *open_uniform_bspline()*
- *closed_uniform_bspline()*
- *rational_bspline_from_arc()*
- *rational_bspline_from_ellipse()*
- *global_bspline_interpolation()*
- *local_cubic_bspline_interpolation()*

Parameters

- **control_points** – iterable of control points as *Vec3* compatible objects
- **order** – spline order (degree + 1)
- **knots** – iterable of knot values
- **weights** – iterable of weight values

property control_points: Sequence[*Vec3*]

Control points as tuple of *Vec3*

property count: int

Count of control points, (n + 1 in text book notation).

property order: int

Order (k) of B-spline = p + 1

property degree: int

Degree (p) of B-spline = order - 1

property max_t: float

Biggest knot value.

property is_rational

Returns `True` if curve is a rational B-spline. (has weights)

property is_clamped

Returns `True` if curve is a clamped (open) B-spline.

knots() → Sequence[float]

Returns a tuple of knot values as floats, the knot vector **always** has order + count values (n + p + 2 in text book notation).

weights() → Sequence[float]

Returns a tuple of weights values as floats, one for each control point or an empty tuple.

params(*segments: int*) → Iterable[float]

Yield evenly spaced parameters for given segment count.

reverse() → *BSpline*

Returns a new `BSpline` object with reversed control point order.

transform(*m:* Matrix44) → *BSpline*

Returns a new `BSpline` object transformed by a `Matrix44` transformation matrix.

approximate(*segments: int = 20*) → Iterable[*Vec3*]

Approximates curve by vertices as `Vec3` objects, vertices count = segments + 1.

flattening(*distance: float*, *segments: int = 4*) → Iterator[*Vec3*]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments between two knots, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the projected curve point onto the segment chord.
- **segments** – minimum segment count between two knots

point(*t: float*) → *Vec3*

Returns point for parameter *t*.

Parameters

t – parameter in range [0, max_t]

points(*t: Iterable[float]*) → Iterable[*Vec3*]

Yields points for parameter vector *t*.

Parameters

t – parameters in range [0, max_t]

derivative(*t: float*, *n: int = 2*) → list[*ezdxf.math._vector.Vec3*]

Return point and derivatives up to *n* <= degree for parameter *t*.

e.g. n=1 returns point and 1st derivative.

Parameters

- **t** – parameter in range [0, max_t]
- **n** – compute all derivatives up to n <= degree

Returns
n+1 values as *Vec3* objects

derivatives(*t: Iterable[float]*, *n: int = 2*) → Iterable[list[*ezdxf.math._vector.Vec3*]]

Yields points and derivatives up to *n* <= degree for parameter vector *t*.

e.g. n=1 returns point and 1st derivative.

Parameters

- **t** – parameters in range [0, max_t]
- **n** – compute all derivatives up to n <= degree

Returns
List of n+1 values as *Vec3* objects

insert_knot(*t: float*) → *BSpline*

Insert an additional knot, without altering the shape of the curve. Returns a new *BSpline* object.

Parameters
t – position of new knot 0 < t < max_t

knot_refinement(*u: Iterable[float]*) → *BSpline*

Insert multiple knots, without altering the shape of the curve. Returns a new *BSpline* object.

Parameters
u – vector of new knots t and for each t: 0 < t < max_t

static from_ellipse(*ellipse:* ConstructionEllipse) → *BSpline*

Returns the ellipse as *BSpline* of 2nd degree with as few control points as possible.

static from_arc(*arc:* ConstructionArc) → *BSpline*

Returns the arc as *BSpline* of 2nd degree with as few control points as possible.

static from_fit_points(*points: Iterable[*UVec*]*, *degree=3*, *method='chord'*) → *BSpline*

Returns *BSpline* defined by fit points.

static arc_approximation(*arc:* ConstructionArc, *num: int = 16*) → *BSpline*

Returns an arc approximation as *BSpline* with *num* control points.

static ellipse_approximation(*ellipse:* ConstructionEllipse, *num: int = 16*) → *BSpline*

Returns an ellipse approximation as *BSpline* with *num* control points.

bezier_decomposition() → Iterable[list[*ezdxf.math._vector.Vec3*]]

Decompose a non-rational B-spline into multiple Bézier curves.

This is the preferred method to represent the most common non-rational B-splines of 3rd degree by cubic Bézier curves, which are often supported by render backends.

Returns
Yields control points of Bézier curves, each Bézier segment has degree+1 control points e.g. B-spline of 3rd degree yields cubic Bézier curves of 4 control points.

cubic_bezier_approximation(*level: int = 3*, *segments: int | None = None*) → Iterable[*Bezier4P*]

Approximate arbitrary B-splines (degree != 3 and/or rational) by multiple segments of cubic Bézier curves. The choice of cubic Bézier curves is based on the widely support of this curves by many render backends. For cubic non-rational B-splines, which is maybe the most common used B-spline, is *bezier_decomposition()* the better choice.

1. approximation by *level*: an educated guess, the first level of approximation segments is based on the count of control points and their distribution along the B-spline, every additional level is a subdivision of the previous level.

E.g. a B-Spline of 8 control points has 7 segments at the first level, 14 at the 2nd level and 28 at the 3rd level, a level >= 3 is recommended.

2. approximation by a given count of evenly distributed approximation segments.

Parameters

- **level** – subdivision level of approximation segments (ignored if argument *segments* is not None)
- **segments** – absolute count of approximation segments

Returns
Yields control points of cubic Bézier curves as *Bezier4P* objects

Bezier

class ezdxf.math.**Bezier**(*defpoints: Iterable[*UVec*]*)

Generic Bézier curve of any degree.

A Bézier curve is a parametric curve used in computer graphics and related fields. Bézier curves are used to model smooth curves that can be scaled indefinitely. "Paths", as they are commonly referred to in image manipulation programs, are combinations of linked Bézier curves. Paths are not bound by the limits of rasterized images and are intuitive to modify. (Source: Wikipedia)

This is a generic implementation which works with any count of definition points greater than 2, but it is a simple and slow implementation. For more performance look at the specialized *Bezier4P* and *Bezier3P* classes.

Objects are immutable.

Parameters
defpoints – iterable of definition points as *Vec3* compatible objects.

control_points

Control points as tuple of *Vec3* objects.

params(*segments: int*) → Iterable[float]

Yield evenly spaced parameters from 0 to 1 for given segment count.

reverse() → *Bezier*

Returns a new Bèzier-curve with reversed control point order.

transform(*m:* Matrix44) → *Bezier*

General transformation interface, returns a new *Bezier* curve.

Parameters
m – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

approximate (*segments: int = 20*) → Iterable[*Vec3*]

Approximates curve by vertices as *Vec3* objects, vertices count = segments + 1.

flattening (*distance: float*, *segments: int = 4*) → Iterable[*Vec3*]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the center of the curve (Cn) to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count

point (*t: float*) → *Vec3*

Returns a point for parameter *t* in range [0, 1] as *Vec3* object.

points (*t: Iterable[float]*) → Iterable[*Vec3*]

Yields multiple points for parameters in vector *t* as *Vec3* objects. Parameters have to be in range [0, 1].

derivative (*t: float*) → tuple[*ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*]

Returns (point, 1st derivative, 2nd derivative) tuple for parameter *t* in range [0, 1] as *Vec3* objects.

derivatives (*t: Iterable[float]*) → Iterable[tuple[*ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*]]

Returns multiple (point, 1st derivative, 2nd derivative) tuples for parameter vector *t* as *Vec3* objects. Parameters in range [0, 1]

Bezier4P

class ezdxf.math.**Bezier4P** (*defpoints: Sequence[*UVec*]*)

Implements an optimized cubic Bézier curve for exact 4 control points.

A Bézier curve is a parametric curve, parameter *t* goes from 0 to 1, where 0 is the first control point and 1 is the fourth control point.

Special behavior:

- 2D control points in, returns 2D results as *Vec2* objects
- 3D control points in, returns 3D results as *Vec3* objects
- Object is immutable.

Parameters
defpoints – iterable of definition points as *Vec2* or *Vec3* compatible objects.

control_points

Control points as tuple of *Vec3* or *Vec2* objects.

reverse () → *Bezier4P*

Returns a new Bèzier-curve with reversed control point order.

transform(*m:* Matrix44) → *Bezier4P*

General transformation interface, returns a new `Bezier4p` curve as a 3D curve.

Parameters
m – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

approximate(*segments: int*) → Iterable[AnyVec]

Approximate Bézier curve by vertices, yields *segments* + 1 vertices as `(x, y[, z])` tuples.

Parameters
segments – count of segments for approximation

flattening(*distance: float*, *segments: int = 4*) → Iterable[*Vec2* | *Vec3*]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the center of the cubic (C3) curve to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count

approximated_length(*segments: int = 128*) → float

Returns estimated length of Bèzier-curve as approximation by line *segments*.

point(*t: float*) → AnyVec

Returns point for location *t*` at the Bèzier-curve.

Parameters
t – curve position in the range `[0, 1]`

tangent(*t: float*) → AnyVec

Returns direction vector of tangent for location *t* at the Bèzier-curve.

Parameters
t – curve position in the range `[0, 1]`

Bezier3P

class ezdxf.math.**Bezier3P**(*defpoints: Sequence[*UVec*]*)

Implements an optimized quadratic Bézier curve for exact 3 control points.

Special behavior:

- 2D control points in, returns 2D results as *Vec2* objects
- 3D control points in, returns 3D results as *Vec3* objects
- Object is immutable.

Parameters
defpoints – iterable of definition points as *Vec2* or *Vec3* compatible objects.

control_points

Control points as tuple of *Vec3* or *Vec2* objects.

reverse() → *Bezier3P*

Returns a new Bèzier-curve with reversed control point order.

transform(*m:* Matrix44) → *Bezier3P*

General transformation interface, returns a new *Bezier3P* curve and it is always a 3D curve.

Parameters
m – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

approximate(*segments: int*) → Iterable['AnyVec']

Approximate Bézier curve by vertices, yields *segments* + 1 vertices as `(x, y[, z])` tuples.

Parameters
segments – count of segments for approximation

flattening(*distance: float*, *segments: int = 4*) → Iterable['AnyVec']

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the center of the quadratic (C2) curve to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count

approximated_length(*segments: int = 128*) → float

Returns estimated length of Bèzier-curve as approximation by line *segments*.

point(*t: float*) → AnyVec

Returns point for location *t*` at the Bèzier-curve.

Parameters
t – curve position in the range `[0, 1]`

tangent(*t: float*) → AnyVec

Returns direction vector of tangent for location *t* at the Bèzier-curve.

Parameters
t – curve position in the range `[0, 1]`

ApproxParamT

class ezdxf.math.**ApproxParamT**(*curve*, *, *max_t: float = 1.0*, *segments: int = 100*)

Approximation tool for parametrized curves.

- approximate parameter *t* for a given distance from the start of the curve
- approximate the distance for a given parameter *t* from the start of the curve

These approximations can be applied to all parametrized curves which provide a `point()` method, like *Bezier4P*, *Bezier3P* and *BSpline*.

The approximation is based on equally spaced parameters from 0 to *max_t* for a given segment count. The `flattening()` method can not be used for the curve approximation, because the required parameter *t* is not logged by the flattening process.

Parameters

- **curve** – curve object, requires a method `point()`
- **max_t** – the max. parameter value
- **segments** – count of approximation segments

property max_t: float

property polyline: *ConstructionPolyline*

param_t(*distance: float*)

Approximate parameter t for the given *distance* from the start of the curve.

distance(*t: float*) → float

Approximate the distance from the start of the curve to the point *t* on the curve.

BezierSurface

class ezdxf.math.**BezierSurface**(*defpoints: list[list[*UVec*]]*)

BezierSurface defines a mesh of *m* x *n* control points. This is a parametric surface, which means the *m*-dimension goes from `0` to `1` as parameter *u* and the *n*-dimension goes from `0` to `1` as parameter *v*.

Parameters

defpoints – matrix (list of lists) of *m* rows and *n* columns: [[m1n1, m1n2, …], [m2n1, m2n2, …] …] each element is a 3D location as `(x, y, z)` tuple.

nrows

count of rows (m-dimension)

ncols

count of columns (n-dimension)

point(*u: float*, *v: float*) → *Vec3*

Returns a point for location (*u*, *v*) at the Bézier surface as `(x, y, z)` tuple, parameters *u* and *v* in the range of `[0, 1]`.

approximate(*usegs: int*, *vsegs: int*) → list[list[*ezdxf.math._vector.Vec3*]]

Approximate surface as grid of `(x, y, z)` *Vec3*.

Parameters

- **usegs** – count of segments in *u*-direction (m-dimension)
- **vsegs** – count of segments in *v*-direction (n-dimension)

Returns

list of *usegs* + 1 rows, each row is a list of *vsegs* + 1 vertices as *Vec3*.

EulerSpiral

class ezdxf.math.**EulerSpiral**(*curvature: float = 1.0*)

This class represents an euler spiral (clothoid) for *curvature* (Radius of curvature).

This is a parametric curve, which always starts at the origin = `(0, 0)`.

Parameters

curvature – radius of curvature

radius (*t: float*) → float

Get radius of circle at distance *t*.

tangent (*t: float*) → *Vec3*

Get tangent at distance *t* as `Vec3` object.

distance (*radius: float*) → float

Get distance L from origin for *radius*.

point (*t: float*) → *Vec3*

Get point at distance *t* as `Vec3`.

circle_center (*t: float*) → *Vec3*

Get circle center at distance *t*.

approximate (*length: float*, *segments: int*) → Iterable[*Vec3*]

Approximate curve of length with line segments. Generates segments+1 vertices as `Vec3` objects.

bspline (*length: float*, *segments: int = 10*, *degree: int = 3*, *method: str = 'uniform'*) → *BSpline*

Approximate euler spiral as B-spline.

Parameters

- **length** – length of euler spiral
- **segments** – count of fit points for B-spline calculation
- **degree** – degree of BSpline
- **method** – calculation method for parameter vector t

Returns
`BSpline`

Clipping

Clipping module: `ezdxf.math.clipping`

ezdxf.math.clipping.**greiner_hormann_union** (*p1: Iterable[*UVec*]*, *p2: Iterable[*UVec*]*) → list[list[*ezdxf.math._vector.Vec2*]]

Returns the UNION of polygon *p1* | polygon *p2*. This algorithm works only for polygons with real intersection points and line end points on face edges are not considered as such intersection points!

ezdxf.math.clipping.**greiner_hormann_difference** (*p1: Iterable[*UVec*]*, *p2: Iterable[*UVec*]*) → list[list[*ezdxf.math._vector.Vec2*]]

Returns the DIFFERENCE of polygon *p1* - polygon *p2*. This algorithm works only for polygons with real intersection points and line end points on face edges are not considered as such intersection points!

ezdxf.math.clipping.**greiner_hormann_intersection** (*p1: Iterable[*UVec*]*, *p2: Iterable[*UVec*]*) → list[list[*ezdxf.math._vector.Vec2*]]

Returns the INTERSECTION of polygon *p1* & polygon *p2*. This algorithm works only for polygons with real intersection points and line end points on face edges are not considered as such intersection points!

class ezdxf.math.clipping.**ClippingPolygon2d** (*vertices: Iterable[*Vec2*]*, *ccw_check=True*)

The clipping path is an arbitrary polygon.

clip_polygon (*polygon: Iterable[*Vec2*]*) → Sequence[*Vec2*]

Returns the clipped polygon.

clip_polyline(*polyline: Iterable[Vec2]*) → Sequence[Sequence[*Vec2*]]

Returns the parts of the clipped polyline.

clip_line(*start:* Vec2, *end:* Vec2) → Sequence[*Vec2*]

Returns the clipped line.

is_inside(*point:* Vec2) → bool

Returns `True` if *point* is inside the clipping polygon.

class ezdxf.math.clipping.**ClippingRect2d**(*bottom_left:* Vec2, *top_right:* Vec2)

The clipping path is a rectangle parallel to the x- and y-axis.

This class will get an optimized implementation in the future.

clip_polygon(*polygon: Iterable[Vec2]*) → Sequence[*Vec2*]

Returns the clipped polygon.

clip_polyline(*polyline: Iterable[Vec2]*) → Sequence[Sequence[*Vec2*]]

Returns the parts of the clipped polyline.

clip_line(*start:* Vec2, *end:* Vec2) → Sequence[*Vec2*]

Returns the clipped line.

is_inside(*point:* Vec2) → bool

Returns `True` if *point* is inside the clipping rectangle.

Clustering

Clustering module: *ezdxf.math.clustering*

ezdxf.math.clustering.**average_cluster_radius**(*clusters: list[list[Union[*ezdxf.math._vector.Vec2, ezdxf.math._vector.Vec3*]]]*) → float

Returns the average cluster radius.

ezdxf.math.clustering.**average_intra_cluster_distance**(*clusters: list[list[Union[*ezdxf.math._vector.Vec2, ezdxf.math._vector.Vec3*]]]*) → float

Returns the average point-to-point intra cluster distance.

ezdxf.math.clustering.**dbscan**(*points: list[Union[*ezdxf.math._vector.Vec2, ezdxf.math._vector.Vec3*]]*, *, *radius: float*, *min_points: int = 4*, *rtree:* RTree | *None = None*, *max_node_size: int = 5*) → list[list[Union[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec3*]]]

DBSCAN clustering.

https://en.wikipedia.org/wiki/DBSCAN

Parameters

- **points** – list of points to cluster
- **radius** – radius of the dense regions
- **min_points** – minimum number of points that needs to be within the *radius* for a point to be a core point (must be >= 2)
- **rtree** – optional *RTree*
- **max_node_size** – max node size for internally created RTree

Returns
list of clusters, each cluster is a list of points

ezdxf.math.clustering.**k_means**(*points: list[Union[*ezdxf.math._vector.Vec2*,* ezdxf.math._vector.Vec3*]]*, *k: int*, *max_iter: int = 10*) → list[list[Union[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec3*]]]

K-means clustering.

https://en.wikipedia.org/wiki/K-means_clustering

Parameters

- **points** – list of points to cluster
- **k** – number of clusters
- **max_iter** – max iterations

Returns
list of clusters, each cluster is a list of points

Linear Algebra

Linear algebra module **for internal usage**: *ezdxf.math.linalg*

Functions

ezdxf.math.linalg.**gauss_jordan_solver**(*A: Iterable[Iterable[float]]*, *B: Iterable[Iterable[float]]*) → tuple[*ezdxf.math.linalg.Matrix*, *ezdxf.math.linalg.Matrix*]

Solves the linear equation system given by a nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B by the Gauss-Jordan algorithm, which is the slowest of all, but it is very reliable. Returns a copy of the modified input matrix *A* and the result matrix *x*.

Internally used for matrix inverse calculation.

Parameters

- **A** – matrix [[a11, a12, …, a1n], [a21, a22, …, a2n], [a21, a22, …, a2n], … [an1, an2, …, ann]]
- **B** – matrix [[b11, b12, …, b1m], [b21, b22, …, b2m], … [bn1, bn2, …, bnm]]

Returns
2-tuple of *Matrix* objects

Raises
ZeroDivisionError – singular matrix

ezdxf.math.linalg.**gauss_jordan_inverse**(*A: Iterable[Iterable[float]]*) → *Matrix*

Returns the inverse of matrix *A* as *Matrix* object.

Hint: For small matrices (n<10) is this function faster than LUDecomposition(m).inverse() and as fast even if the decomposition is already done.

Raises
ZeroDivisionError – singular matrix

ezdxf.math.linalg.**gauss_vector_solver**(*A: Iterable[Iterable[float]]*, *B: Iterable[float]*) → list[float]

Solves the linear equation system given by a nxn Matrix A . x = B, right-hand side quantities as vector B with n elements by the Gauss-Elimination algorithm, which is faster than the Gauss-Jordan algorithm. The speed improvement is more significant for solving multiple right-hand side quantities as matrix at once.

Reference implementation for error checking.

Parameters

- **A** – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]
- **B** – vector [b1, b2, ..., bn]

Returns
vector as list of floats

Raises
ZeroDivisionError – singular matrix

ezdxf.math.linalg.**gauss_matrix_solver**(*A: Iterable[Iterable[float]]*, *B: Iterable[Iterable[float]]*) → *Matrix*

Solves the linear equation system given by a nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B by the Gauss-Elimination algorithm, which is faster than the Gauss-Jordan algorithm.

Reference implementation for error checking.

Parameters

- **A** – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]
- **B** – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns
matrix as *Matrix* object

Raises
ZeroDivisionError – singular matrix

ezdxf.math.linalg.**tridiagonal_vector_solver**(*A: Iterable[Iterable[float]]*, *B: Iterable[float]*) → list[float]

Solves the linear equation system given by a tri-diagonal nxn Matrix A . x = B, right-hand side quantities as vector B. Matrix A is diagonal matrix defined by 3 diagonals [-1 (a), 0 (b), +1 (c)].

Note: a0 is not used but has to be present, cn-1 is also not used and must not be present.

If an `ZeroDivisionError` exception occurs, the equation system can possibly be solved by `BandedMatrixLU(A, 1, 1).solve_vector(B)`

Parameters

- **A** – diagonal matrix [[a0..an-1], [b0..bn-1], [c0..cn-1]]

```
[[b0, c0, 0, 0, ...],
[a1, b1, c1, 0, ...],
[0, a2, b2, c2, ...],
... ]
```

- **B** – iterable of floats [[b1, b1, ..., bn]

Returns
list of floats

Raises
ZeroDivisionError – singular matrix

ezdxf.math.linalg.**tridiagonal_matrix_solver**(*A: Iterable[Iterable[float]]*, *B: Iterable[Iterable[float]]*) → *Matrix*

Solves the linear equation system given by a tri-diagonal nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B. Matrix A is diagonal matrix defined by 3 diagonals [-1 (a), 0 (b), +1 (c)].

Note: a0 is not used but has to be present, cn-1 is also not used and must not be present.

If an `ZeroDivisionError` exception occurs, the equation system can possibly be solved by `BandedMatrixLU(A, 1, 1).solve_vector(B)`

Parameters

- **A** – diagonal matrix [[a0..an-1], [b0..bn-1], [c0..cn-1]]

```
[[b0, c0, 0, 0, ...],
[a1, b1, c1, 0, ...],
[0, a2, b2, c2, ...],
... ]
```

- **B** – matrix [[b11, b12, …, b1m], [b21, b22, …, b2m], … [bn1, bn2, …, bnm]]

Returns
matrix as *Matrix* object

Raises
ZeroDivisionError – singular matrix

ezdxf.math.linalg.**banded_matrix**(*A:* Matrix, *check_all=True*) → tuple[*ezdxf.math.linalg.Matrix*, int, int]

Transform matrix A into a compact banded matrix representation. Returns compact representation as *Matrix* object and lower- and upper band count m1 and m2.

Parameters

- **A** – input *Matrix*
- **check_all** – check all diagonals if `True` or abort testing after first all zero diagonal if `False`.

ezdxf.math.linalg.**detect_banded_matrix**(*A:* Matrix, *check_all=True*) → tuple[int, int]

Returns lower- and upper band count m1 and m2.

Parameters

- **A** – input *Matrix*
- **check_all** – check all diagonals if `True` or abort testing after first all zero diagonal if `False`.

ezdxf.math.linalg.**compact_banded_matrix**(*A:* Matrix, *m1: int*, *m2: int*) → *Matrix*

Returns compact banded matrix representation as *Matrix* object.

Parameters

- **A** – matrix to transform
- **m1** – lower band count, excluding main matrix diagonal
- **m2** – upper band count, excluding main matrix diagonal

ezdxf.math.linalg.**freeze_matrix**(*A: Iterable[Iterable[float]]* | Matrix) → *Matrix*

Returns a frozen matrix, all data is stored in immutable tuples.

Matrix Class

class ezdxf.math.linalg.**Matrix**(*items: Any = None*, *shape: Tuple[int, int] | None = None*, *matrix: List[List[float]] | None = None*)

Basic matrix implementation without any optimization for speed or memory usage. Matrix data is stored in row major order, this means in a list of rows, where each row is a list of floats. Direct access to the data is accessible by the attribute `Matrix.matrix`.

The matrix can be frozen by function *freeze_matrix()* or method *Matrix.freeze()*, than the data is stored in immutable tuples.

Initialization:

- Matrix(shape=(rows, cols)) … new matrix filled with zeros
- Matrix(matrix[, shape=(rows, cols)]) … from copy of matrix and optional reshape
- Matrix([[row_0], [row_1], …, [row_n]]) … from Iterable[Iterable[float]]
- Matrix([a1, a2, …, an], shape=(rows, cols)) … from Iterable[float] and shape

nrows

Count of matrix rows.

ncols

Count of matrix columns.

shape

Shape of matrix as (n, m) tuple for n rows and m columns.

static reshape(*items: Iterable[float]*, *shape: Tuple[int, int]*) → *Matrix*

Returns a new matrix for iterable *items* in the configuration of *shape*.

classmethod identity(*shape: Tuple[int, int]*) → *Matrix*

Returns the identity matrix for configuration *shape*.

row(*index: int*) → list[float]

Returns row *index* as list of floats.

iter_row(*index: int*) → Iterator[float]

Yield values of row *index*.

col(*index: int*) → list[float]

Return column *index* as list of floats.

iter_col(*index: int*) → Iterator[float]

Yield values of column *index*.

diag(*index: int*) → list[float]

Returns diagonal *index* as list of floats.

An *index* of 0 specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.

e.g. given a 4x4 matrix:

- index 0 is [00, 11, 22, 33],
- index -1 is [10, 21, 32] and
- index +1 is [01, 12, 23]

iter_diag(*index: int*) → Iterator[float]

Yield values of diagonal *index*, see also *diag()*.

rows() → List[List[float]]

Return a list of all rows.

cols() → List[List[float]]

Return a list of all columns.

set_row(*index: int*, *items: float | Sequence[float] = 1.0*) → None

Set row values to a fixed value or from an iterable of floats.

set_col(*index: int*, *items: float | Iterable[float] = 1.0*) → None

Set column values to a fixed value or from an iterable of floats.

set_diag(*index: int = 0*, *items: float | Iterable[float] = 1.0*) → None

Set diagonal values to a fixed value or from an iterable of floats.

An *index* of `0` specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.

e.g. given a 4x4 matrix: index `0` is [00, 11, 22, 33], index `-1` is [10, 21, 32] and index `+1` is [01, 12, 23]

append_row(*items: Sequence[float]*) → None

Append a row to the matrix.

append_col(*items: Sequence[float]*) → None

Append a column to the matrix.

swap_rows(*a: int*, *b: int*) → None

Swap rows *a* and *b* inplace.

swap_cols(*a: int*, *b: int*) → None

Swap columns *a* and *b* inplace.

transpose() → *Matrix*

Returns a new transposed matrix.

inverse() → *Matrix*

Returns inverse of matrix as new object.

determinant() → float

Returns determinant of matrix, raises `ZeroDivisionError` if matrix is singular.

freeze() → *Matrix*

Returns a frozen matrix, all data is stored in immutable tuples.

lu_decomp() → *LUDecomposition*

Returns the LU decomposition as *LUDecomposition* object, a faster linear equation solver.

__getitem__(*item: tuple[int, int]*) → float

Get value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

__setitem__(*item: tuple[int, int]*, *value: float*)

Set value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

__eq__(*other: object*) → bool

Returns `True` if matrices are equal, tolerance value for comparison is adjustable by the attribute `Matrix.abs_tol`.

__add__(*other:* Matrix | *float*) → *Matrix*

Matrix addition by another matrix or a float, returns a new matrix.

__sub__(*other:* Matrix | *float*) → *Matrix*

Matrix subtraction by another matrix or a float, returns a new matrix.

__mul__(*other:* Matrix | *float*) → *Matrix*

Matrix multiplication by another matrix or a float, returns a new matrix.

LUDecomposition Class

class ezdxf.math.linalg.**LUDecomposition**(*A: Iterable[Iterable[float]]*)

Represents a LU decomposition matrix of A, raise `ZeroDivisionError` for a singular matrix.

This algorithm is a little bit faster than the Gauss-Elimination algorithm using CPython and much faster when using pypy.

The `LUDecomposition.matrix` attribute gives access to the matrix data as list of rows like in the *Matrix* class, and the `LUDecomposition.index` attribute gives access to the swapped row indices.

Parameters
A – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]

Raises
ZeroDivisionError – singular matrix

nrows

Count of matrix rows (and cols).

solve_vector(*B: Iterable[float]*) → list[float]

Solves the linear equation system given by the nxn Matrix A . x = B, right-hand side quantities as vector B with n elements.

Parameters
B – vector [b1, b2, ..., bn]

Returns
vector as list of floats

solve_matrix(*B: Iterable[Iterable[float]]*) → *Matrix*

Solves the linear equation system given by the nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B.

Parameters
B – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns
matrix as *Matrix* object

inverse() → *Matrix*

Returns the inverse of matrix as *Matrix* object, raise `ZeroDivisionError` for a singular matrix.

determinant() → float

Returns the determinant of matrix, raises `ZeroDivisionError` if matrix is singular.

BandedMatrixLU Class

class ezdxf.math.linalg.**BandedMatrixLU**(*A:* Matrix, *m1: int*, *m2: int*)

Represents a LU decomposition of a compact banded matrix.

upper

Upper triangle

lower

Lower triangle

m1

Lower band count, excluding main matrix diagonal

m2

Upper band count, excluding main matrix diagonal

index

Swapped indices

nrows

Count of matrix rows.

solve_vector(*B: Iterable[float]*) → list[float]

Solves the linear equation system given by the banded nxn Matrix A . x = B, right-hand side quantities as vector B with n elements.

Parameters
B – vector [b1, b2, …, bn]

Returns
vector as list of floats

solve_matrix(*B: Iterable[Iterable[float]]*) → *Matrix*

Solves the linear equation system given by the banded nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B.

Parameters
B – matrix [[b11, b12, …, b1m], [b21, b22, …, b2m], … [bn1, bn2, …, bnm]]

Returns
matrix as *Matrix* object

determinant() → float

Returns the determinant of matrix.

RTree

RTree module: *ezdxf.math.rtree*

class ezdxf.math.rtree.**RTree**(*points: Iterable[AnyVec]*, *max_node_size: int = 5*)

Immutable spatial search tree loosely based on R-trees.

The search tree is buildup once at initialization and immutable afterwards, because rebuilding the tree after inserting or deleting nodes is very costly and also keeps the implementation very simple. Without the ability to alter the content the restrictions which forces the tree balance at growing and shrinking of the original R-trees, could be ignored, like the fixed minimum and maximum node size.

This class uses internally only 3D bounding boxes, but also supports `Vec2` as well as `Vec3` objects as input data, but point types should not be mixed in a single search tree.

The point objects keep their type and identity and the returned points of queries can be compared by the `is` operator for identity to the input points.

The implementation requires a maximum node size of at least 2 and does not support empty trees!

Raises
ValueError – max. node size too small or no data given

__len__()

Returns the count of points in the search tree.

__iter__() → Iterator[*Vec2* | *Vec3*]

Yields all points in the search tree.

contains(*point:* Vec2 | Vec3) → bool

Returns `True` if *point* exists, the comparison is done by the `isclose()` method and not by the identity operator `is`.

nearest_neighbor(*target:* Vec2 | Vec3) → tuple[Union[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec3*], float]

Returns the closest point to the *target* point and the distance between these points.

points_in_sphere(*center:* Vec2 | Vec3, *radius: float*) → Iterator[*Vec2* | *Vec3*]

Returns all points in the range of the given sphere including the points at the boundary.

points_in_bbox(*bbox:* BoundingBox) → Iterator[*Vec2* | *Vec3*]

Returns all points in the range of the given bounding box including the points at the boundary.

avg_leaf_size(*spread: float = 1.0*) → float

Returns the average size of the leaf bounding boxes. The size of a leaf bounding box is the maximum size in all dimensions. Excludes outliers of sizes beyond mean + standard deviation * spread. Returns 0.0 if less than two points in tree.

avg_spherical_envelope_radius(*spread: float = 1.0*) → float

Returns the average radius of spherical envelopes of the leaf nodes. Excludes outliers with radius beyond mean + standard deviation * spread. Returns 0.0 if less than two points in tree.

avg_nn_distance(*spread: float = 1.0*) → float

Returns the average of the nearest neighbor distances inside (!) leaf nodes. Excludes outliers with a distance beyond the overall mean + standard deviation * spread. Returns 0.0 if less than two points in tree.

> **Warning:** This is a brute force check with O(n!) for each leaf node, where n is the point count of the leaf node.

Triangulation

Triangulation module: *ezdxf.math.triangulation*

ezdxf.math.triangulation.**mapbox_earcut_2d**(*exterior: Iterable[*UVec*]*, *holes: Iterable[Iterable[*UVec*]] | None = None*) → list[Sequence[*ezdxf.math._vector.Vec2*]]

Mapbox triangulation algorithm with hole support for 2D polygons.

Implements a modified ear slicing algorithm, optimized by z-order curve hashing and extended to handle holes, twisted polygons, degeneracies and self-intersections in a way that doesn't guarantee correctness of triangulation, but attempts to always produce acceptable results for practical data.

Source: https://github.com/mapbox/earcut

Parameters

- **exterior** – exterior polygon as iterable of `Vec2` objects
- **holes** – iterable of holes as iterable of `Vec2` objects, a hole with single point represents a Steiner point.

Returns
yields the result as 3-tuples of `Vec2` objects

ezdxf.math.triangulation.**mapbox_earcut_3d**(*exterior: Iterable[*UVec*]*, *holes: Iterable[Iterable[*UVec*]] | None = None*) → Iterator[tuple[*ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*]]

Mapbox triangulation algorithm with hole support for flat 3D polygons.

Implements a modified ear slicing algorithm, optimized by z-order curve hashing and extended to handle holes, twisted polygons, degeneracies and self-intersections in a way that doesn't guarantee correctness of triangulation, but attempts to always produce acceptable results for practical data.

Source: https://github.com/mapbox/earcut

Parameters

- **exterior** – exterior polygon as iterable of `Vec3` objects
- **holes** – iterable of holes as iterable of `Vec3` objects, a hole with single point represents a Steiner point.

Returns
yields the result as 3-tuples of `Vec3` objects

Raises

- **TypeError** – invalid input data type
- **ZeroDivisionError** – normal vector calculation failed

6.9.8 Construction

Path

This module implements a geometric `Path`, supported by several render backends, with the goal to create such paths from DXF entities like LWPOLYLINE, POLYLINE or HATCH and send them to the render backend, see `ezdxf.addons.drawing`.

Minimum common interface:

- **matplotlib: PathPatch**
  - matplotlib.path.Path() codes:
  - MOVETO
  - LINETO
  - CURVE3 - quadratic Bèzier-curve
  - CURVE4 - cubic Bèzier-curve
- **PyQt: QPainterPath**
  - moveTo()
  - lineTo()
  - quadTo() - quadratic Bèzier-curve (converted to a cubic Bèzier-curve)
  - cubicTo() - cubic Bèzier-curve
- **PyCairo: Context**
  - move_to()
  - line_to()
  - no support for quadratic Bèzier-curve
  - curve_to() - cubic Bèzier-curve
- **SVG: SVG-Path**
  - "M" - absolute move to
  - "L" - absolute line to
  - "Q" - absolute quadratic Bèzier-curve
  - "C" - absolute cubic Bèzier-curve

ARC and ELLIPSE entities are approximated by multiple cubic Bézier-curves, which are close enough for display rendering. Non-rational SPLINES of 3rd degree can be represented exact as multiple cubic Bézier-curves, other B-splines will be approximated. The XLINE and the RAY entities are not supported, because of their infinite nature.

This `Path` class is a full featured 3D object, although the backends only support 2D paths.

Hint: A `Path` can not represent a point. A `Path` with only a start point yields no vertices!

The usability of the `Path` class expanded by the introduction of the reverse conversion from `Path` to DXF entities (LWPOLYLINE, POLYLINE, LINE), and many other tools in *ezdxf* v0.16. To emphasize this new usability, the `Path` class has got its own subpackage `ezdxf.path`.

Empty-Path
Contains only a start point, the length of the path is 0 and the methods `Path.approximate()`, `Path.flattening()` and `Path.control_vertices()` do not yield any vertices.

Single-Path
The `Path` object contains only one path without gaps, the property `Path.has_sub_paths` is `False` and the method `Path.sub_paths()` yields only this one path.

Multi-Path
The `Path` object contains more than one path, the property `Path.has_sub_paths` is `True` and the method `Path.sub_paths()` yields all paths within this object as single-path objects. It is not possible to detect the orientation of a multi-path object, therefore the methods `Path.has_clockwise_orientation()`, `Path.clockwise()` and `Path.counter_clockwise()` raise a `TypeError` exception.

Warning: Always import from the top level `ezdxf.path`, never from the sub-modules

Factory Functions

Functions to create `Path` objects from other objects.

`ezdxf.path.`**`make_path`**(*entity:* DXFEntity) → *Path*

Factory function to create a single `Path` object from a DXF entity. Supported DXF types:

- LINE
- CIRCLE
- ARC
- ELLIPSE
- SPLINE and HELIX
- LWPOLYLINE
- 2D and 3D POLYLINE
- SOLID, TRACE, 3DFACE
- IMAGE, WIPEOUT clipping path
- VIEWPORT clipping path
- HATCH as *Multi-Path* object

Parameters

- **`entity`** – DXF entity
- **`segments`** – minimal count of cubic Bézier-curves for elliptical arcs like CIRCLE, ARC, ELLIPSE, BULGE see `Path.add_ellipse()`
- **`level`** – subdivide level for SPLINE approximation, see `Path.add_spline()`

Raises
`TypeError` – for unsupported DXF types

`ezdxf.path.`**`from_hatch`**(*hatch: DXFPolygon*, *offset:* Vec3 = *NULLVEC*) → Iterator[Path]

Yield all HATCH/MPOLYGON boundary paths as separated `Path` objects in WCS coordinates.

ezdxf.path.**from_vertices**(*vertices: Iterable[UVec]*, *close=False*) → Path

Returns a *Path* object from the given *vertices*.

ezdxf.path.**from_matplotlib_path**(*mpath*, *curves=True*) → Iterator[Path]

Yields multiple *Path* objects from a Matplotlib Path (TextPath) object. (requires Matplotlib)

ezdxf.path.**multi_path_from_matplotlib_path**(*mpath*, *curves=True*) → Path

Returns a *Path* object from a Matplotlib Path (TextPath) object. (requires Matplotlib). Returns a multi-path object if necessary.

ezdxf.path.**from_qpainter_path**(*qpath*) → Iterator[Path]

Yields multiple *Path* objects from a QPainterPath. (requires Qt bindings)

ezdxf.path.**multi_path_from_qpainter_path**(*qpath*) → Path

Returns a *Path* objects from a QPainterPath. Returns a multi-path object if necessary. (requires Qt bindings)

Render Functions

Functions to create DXF entities from paths and add them to the modelspace, a paperspace layout or a block definition.

ezdxf.path.**render_hatches**(*layout: GenericLayoutType*, *paths: Iterable[Path]*, ***, *edge_path: bool = True*, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *g1_tol: float = G1_TOL*, *extrusion: UVec = Z_AXIS*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as *Hatch* entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **edge_path** – `True` for edge paths build of LINE and SPLINE edges, `False` for only LWPOLYLINE paths as boundary paths
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve to flatten polyline paths
- **g1_tol** – tolerance for G1 continuity check to separate SPLINE edges
- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns

created entities in an *EntityQuery* object

ezdxf.path.**render_lines**(*layout: GenericLayoutType*, *paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as *Line* entities.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*

- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns
created entities in an *EntityQuery* object

ezdxf.path.**render_lwpolylines**(*layout: GenericLayoutType*, *paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as *LWPolyline* entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve
- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns
created entities in an *EntityQuery* object

ezdxf.path.**render_mpolygons**(*layout: GenericLayoutType*, *paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as *MPolygon* entities. The MPOLYGON entity supports only polyline boundary paths. All curves will be approximated.

The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve to flatten polyline paths
- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns
created entities in an *EntityQuery* object

ezdxf.path.**render_polylines2d**(*layout: GenericLayoutType*, *paths: Iterable[Path]*, ***, *distance: float = 0.01*, *segments: int = 4*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as 2D *Polyline* entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector.The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve
- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns
created entities in an *EntityQuery* object

ezdxf.path.**render_polylines3d**(*layout: GenericLayoutType*, *paths: Iterable[*Path*]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as 3D *Polyline* entities.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns
created entities in an *EntityQuery* object

ezdxf.path.**render_splines_and_polylines**(*layout: GenericLayoutType*, *paths: Iterable[*Path*]*, ***, *g1_tol: float = G1_TOL*, *dxfattribs=None*) → *EntityQuery*

Render the given *paths* into *layout* as *Spline* and 3D *Polyline* entities.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of *Path* objects
- **g1_tol** – tolerance for G1 continuity check
- **dxfattribs** – additional DXF attribs

Returns
created entities in an *EntityQuery* object

Entity Maker

Functions to create DXF entities from paths.

ezdxf.path.**to_hatches**(*paths: Iterable[Path]*, ***, *edge_path: bool = True*, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *g1_tol: float = G1_TOL*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → Iterator[Hatch]

Convert the given *paths* into `Hatch` entities. Uses LWPOLYLINE paths for boundaries without curves and edge paths, build of LINE and SPLINE edges, as boundary paths for boundaries including curves. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **paths** – iterable of `Path` objects
- **edge_path** – `True` for edge paths build of LINE and SPLINE edges, `False` for only LWPOLYLINE paths as boundary paths
- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve to flatten LWPOLYLINE paths
- **g1_tol** – tolerance for G1 continuity check to separate SPLINE edges
- **extrusion** – extrusion vector to all paths
- **dxfattribs** – additional DXF attribs

Returns
iterable of `Hatch` objects

ezdxf.path.**to_lines**(*paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *dxfattribs=None*) → Iterator[Line]

Convert the given *paths* into `Line` entities.

Parameters

- **paths** – iterable of `Path` objects
- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns
iterable of `Line` objects

ezdxf.path.**to_lwpolylines**(*paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → Iterator[LWPolyline]

Convert the given *paths* into `LWPolyline` entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **paths** – iterable of `Path` objects
- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve

- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns
iterable of *LWPolyline* objects

ezdxf.path.**to_mpolygons**(*paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → Iterator[MPolygon]

Convert the given *paths* into *MPolygon* entities. In contrast to HATCH, MPOLYGON supports only polyline boundary paths. All curves will be approximated.

The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve to flatten LWPOLYLINE paths
- **extrusion** – extrusion vector to all paths
- **dxfattribs** – additional DXF attribs

Returns
iterable of *MPolygon* objects

ezdxf.path.**to_polylines2d**(*paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *extrusion:* UVec *= Z_AXIS*, *dxfattribs=None*) → Iterator[Polyline]

Convert the given *paths* into 2D *Polyline* entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve
- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns
iterable of 2D *Polyline* objects

ezdxf.path.**to_polylines3d**(*paths: Iterable[Path]*, ***, *distance: float = MAX_DISTANCE*, *segments: int = MIN_SEGMENTS*, *dxfattribs=None*) → Iterator[Polyline]

Convert the given *paths* into 3D *Polyline* entities.

Parameters

- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve

- **dxfattribs** – additional DXF attribs

Returns
iterable of 3D *Polyline* objects

ezdxf.path.**to_splines_and_polylines**(*paths: Iterable[Path]*, *, *g1_tol: float = G1_TOL*, *dxfattribs=None*) → Iterator[Spline | Polyline]

Convert the given *paths* into *Spline* and 3D *Polyline* entities.

Parameters

- **paths** – iterable of *Path* objects
- **g1_tol** – tolerance for G1 continuity check
- **dxfattribs** – additional DXF attribs

Returns
iterable of *Line* objects

Tool Maker

Functions to create construction tools.

ezdxf.path.**to_bsplines_and_vertices**(*path: Path*, *g1_tol: float = G1_TOL*) → Iterator[*BSpline* | List[*Vec3*]]

Convert a *Path* object into multiple cubic B-splines and polylines as lists of vertices. Breaks adjacent Bèzier without G1 continuity into separated B-splines.

Parameters

- **path** – *Path* objects
- **g1_tol** – tolerance for G1 continuity check

Returns
BSpline and lists of *Vec3*

ezdxf.path.**to_matplotlib_path**(*paths: Iterable[Path]*, *extrusion:* UVec = *Z_AXIS*)

Convert the given *paths* into a single `matplotlib.path.Path` object. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector.The default extrusion vector is the WCS z-axis. The Matplotlib *Path* is a 2D object with *OCS* coordinates and the z-elevation is lost. (requires Matplotlib)

Parameters

- **paths** – iterable of *Path* objects
- **extrusion** – extrusion vector for all paths

Returns
matplotlib Path in OCS!

ezdxf.path.**to_qpainter_path**(*paths: Iterable[Path]*, *extrusion:* UVec = *Z_AXIS*)

Convert the given *paths* into a `QtGui.QPainterPath` object. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The `QPainterPath` is a 2D object with *OCS* coordinates and the z-elevation is lost. (requires Qt bindings)

Parameters

- **paths** – iterable of *Path* objects

- **extrusion** – extrusion vector for all paths

Returns
QPainterPath in OCS!

Utility Functions

ezdxf.path.**add_bezier3p**(*path: Path*, *curves: Iterable[*Bezier3P*]*) → None

Add multiple quadratic Bèzier-curves to the given *path*.

Auto-detect the connection point to the given *path*, if neither the start- nor the end point of the curves is close to the path end point, a line from the path end point to the start point of the first curve will be added automatically.

ezdxf.path.**add_bezier4p**(*path: Path*, *curves: Iterable[*Bezier4P*]*) → None

Add multiple cubic Bèzier-curves to the given *path*.

Auto-detect the connection point to the given *path*, if neither the start- nor the end point of the curves is close to the path end point, a line from the path end point to the start point of the first curve will be added automatically.

ezdxf.path.**add_ellipse**(*path: Path*, *ellipse:* ConstructionEllipse, *segments=1*, *reset=True*) → None

Add an elliptical arc as multiple cubic Bèzier-curves to the given *path*, use *from_arc()* constructor of class *ConstructionEllipse* to add circular arcs.

Auto-detect the connection point to the given *path*, if neither the start- nor the end point of the ellipse is close to the path end point, a line from the path end point to the ellipse start point will be added automatically (see *add_bezier4p()*).

By default, the start of an **empty** path is set to the start point of the ellipse, setting argument *reset* to `False` prevents this behavior.

Parameters

- **path** – *Path* object
- **ellipse** – ellipse parameters as *ConstructionEllipse* object
- **segments** – count of Bèzier-curve segments, at least one segment for each quarter (pi/2), `1` for as few as possible.
- **reset** – set start point to start of ellipse if path is empty

ezdxf.path.**add_spline**(*path: Path*, *spline:* BSpline, *level=4*, *reset=True*) → None

Add a B-spline as multiple cubic Bèzier-curves.

Non-rational B-splines of 3rd degree gets a perfect conversion to cubic Bézier curves with a minimal count of curve segments, all other B-spline require much more curve segments for approximation.

Auto-detect the connection point to the given *path*, if neither the start- nor the end point of the B-spline is close to the path end point, a line from the path end point to the start point of the B-spline will be added automatically. (see *add_bezier4p()*).

By default, the start of an **empty** path is set to the start point of the spline, setting argument *reset* to `False` prevents this behavior.

Parameters

- **path** – *Path* object
- **spline** – B-spline parameters as *BSpline* object
- **level** – subdivision level of approximation segments

- **reset** – set start point to start of spline if path is empty

ezdxf.path.**bbox**(*paths: Iterable[Path]*, *, *fast=False*) → *BoundingBox*

Returns the *BoundingBox* for the given paths.

Parameters

- **paths** – iterable of *Path* objects
- **fast** – calculates the precise bounding box of Bèzier curves if `False`, otherwise uses the control points of Bézier curves to determine their bounding box.

ezdxf.path.**chamfer**(*points: Sequence[*Vec3*]*, *length: float*) → Path

Returns a *Path* with chamfers of given *length* between straight line segments.

Parameters

- **points** – coordinates of the line segments
- **length** – chamfer length

ezdxf.path.**chamfer2**(*points: Sequence[*Vec3*]*, *a: float*, *b: float*) → Path

Returns a *Path* with chamfers at the given distances *a* and *b* from the segment points between straight line segments.

Parameters

- **points** – coordinates of the line segments
- **a** – distance of the chamfer start point to the segment point
- **b** – distance of the chamfer end point to the segment point

ezdxf.path.**fillet**(*points: Sequence[*Vec3*]*, *radius: float*) → Path

Returns a *Path* with circular fillets of given *radius* between straight line segments.

Parameters

- **points** – coordinates of the line segments
- **radius** – fillet radius

ezdxf.path.**fit_paths_into_box**(*paths: Iterable[Path]*, *size: tuple[float, float, float]*, *uniform: bool = True*, *source_box:* BoundingBox | *None = None*) → list[ezdxf.path.path.Path]

Scale the given *paths* to fit into a box of the given *size*, so that all path vertices are inside these borders. If *source_box* is `None` the default source bounding box is calculated from the control points of the *paths*.

Note: if the target size has a z-size of 0, the *paths* are projected into the xy-plane, same is true for the x-size, projects into the yz-plane and the y-size, projects into and xz-plane.

Parameters

- **paths** – iterable of *Path* objects
- **size** – target box size as tuple of x-, y- and z-size values
- **uniform** – `True` for uniform scaling
- **source_box** – pass precalculated source bounding box, or `None` to calculate the default source bounding box from the control vertices

ezdxf.path.**have_close_control_vertices**(*a: Path*, *b: Path*, *, *rel_tol=1e-9*, *abs_tol=1e-12*) → bool

Returns `True` if the control vertices of given paths are close.

ezdxf.path.**lines_to_curve3**(*path: Path*) → Path

Replaces all lines by quadratic Bézier curves. Returns a new *Path* instance.

ezdxf.path.**lines_to_curve4**(*path: Path*) → Path

Replaces all lines by cubic Bézier curves. Returns a new *Path* instance.

ezdxf.path.**polygonal_fillet**(*points: Sequence[*Vec3*]*, *radius: float*, *count: int = 32*) → Path

Returns a *Path* with polygonal fillets of given *radius* between straight line segments. The *count* argument defines the vertex count of the fillet for a full circle.

Parameters

- **points** – coordinates of the line segments
- **radius** – fillet radius
- **count** – polygon vertex count for a full circle, minimum is 4

ezdxf.path.**single_paths**(*paths: Iterable[Path]*) → Iterable[Path]

Yields all given paths and their sub-paths as single path objects.

ezdxf.path.**to_multi_path**(*paths: Iterable[Path]*) → Path

Returns a multi-path object from all given paths and their sub-paths. Ignores paths without any commands (empty paths).

ezdxf.path.**transform_paths**(*paths: Iterable[Path]*, *m:* Matrix44) → list[ezdxf.path.path.Path]

Transform multiple *Path* objects at once by transformation matrix *m*. Returns a list of the transformed *Path* objects.

Parameters

- **paths** – iterable of *Path* objects
- **m** – transformation matrix of type *Matrix44*

ezdxf.path.**transform_paths_to_ocs**(*paths: Iterable[Path]*, *ocs:* OCS) → list[ezdxf.path.path.Path]

Transform multiple *Path* objects at once from WCS to OCS. Returns a list of the transformed *Path* objects.

Parameters

- **paths** – iterable of *Path* objects
- **ocs** – OCS transformation of type *OCS*

ezdxf.path.**triangulate**(*paths: Iterable[Path]*, *max_sagitta: float = 0.01*, *min_segments: int = 16*) → Iterator[Sequence[*Vec2*]]

Tessellate nested 2D paths into triangle-faces. For 3D paths the projection onto the xy-plane will be triangulated.

Parameters

- **paths** – iterable of nested Path instances
- **max_sagitta** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.
- **min_segments** – minimum segment count per Bézier curve

Basic Shapes

ezdxf.path.**elliptic_transformation**(*center:* UVec = *(0, 0, 0)*, *radius: float = 1*, *ratio: float = 1*, *rotation: float = 0*) → *Matrix44*

Returns the transformation matrix to transform a unit circle into an arbitrary circular- or elliptic arc.

Example how to create an ellipse with a major axis length of 3, a minor axis length 1.5 and rotated about 90°:

```
m = elliptic_transformation(radius=3, ratio=0.5, rotation=math.pi / 2)
ellipse = shapes.unit_circle(transform=m)
```

Parameters

- **center** – curve center in WCS
- **radius** – radius of the major axis in drawing units
- **ratio** – ratio of minor axis to major axis
- **rotation** – rotation angle about the z-axis in radians

ezdxf.path.**gear**(*count: int*, *top_width: float*, *bottom_width: float*, *height: float*, *outside_radius: float*, *transform:* Matrix44 = *None*) → Path

Returns a gear (cogwheel) shape as a *Path* object, with the center at (0, 0, 0). The base geometry is created by function *ezdxf.render.forms.gear()*.

Warning: This function does not create correct gears for mechanical engineering!

Parameters

- **count** – teeth count >= 3
- **top_width** – teeth width at outside radius
- **bottom_width** – teeth width at base radius
- **height** – teeth height; base radius = outside radius - height
- **outside_radius** – outside radius
- **transform** – transformation Matrix applied to the gear shape

ezdxf.path.**helix**(*radius: float*, *pitch: float*, *turns: float*, *ccw=True*, *segments: int = 4*) → Path

Returns a helix as a *Path* object. The center of the helix is always (0, 0, 0), a positive *pitch* value creates a helix along the +z-axis, a negative value along the -z-axis.

Parameters

- **radius** – helix radius
- **pitch** – the height of one complete helix turn
- **turns** – count of turns
- **ccw** – creates a counter-clockwise turning (right-handed) helix if `True`
- **segments** – cubic Bezier segments per turn

ezdxf.path.**ngon**(*count: int*, *length: float | None = None*, *radius: float = 1.0*, *transform:* Matrix44 *| None = None*) → Path

Returns a regular polygon a `Path` object, with the center at (0, 0, 0). The polygon size is determined by the edge *length* or the circum *radius* argument. If both are given *length* has higher priority. Default size is a *radius* of 1. The ngon starts with the first vertex is on the x-axis! The base geometry is created by function `ezdxf.render.forms.ngon()`.

Parameters

- **count** – count of polygon corners >= 3
- **length** – length of polygon side
- **radius** – circum radius, default is 1
- **transform** – transformation Matrix applied to the ngon

ezdxf.path.**rect**(*width: float = 1*, *height: float = 1*, *transform:* Matrix44 *= None*) → Path

Returns a closed rectangle as a `Path` object, with the center at (0, 0, 0) and the given *width* and *height* in drawing units.

Parameters

- **width** – width of the rectangle in drawing units, width > 0
- **height** – height of the rectangle in drawing units, height > 0
- **transform** – transformation Matrix applied to the rectangle

ezdxf.path.**star**(*count: int*, *r1: float*, *r2: float*, *transform:* Matrix44 *= None*) → Path

Returns a star shape as a `Path` object, with the center at (0, 0, 0).

Argument *count* defines the count of star spikes, *r1* defines the radius of the "outer" vertices and *r2* defines the radius of the "inner" vertices, but this does not mean that *r1* has to be greater than *r2*. The star shape starts with the first vertex is on the x-axis! The base geometry is created by function `ezdxf.render.forms.star()`.

Parameters

- **count** – spike count >= 3
- **r1** – radius 1
- **r2** – radius 2
- **transform** – transformation Matrix applied to the star

ezdxf.path.**unit_circle**(*start_angle: float = 0*, *end_angle: float = math.tau*, *segments: int = 1*, *transform:* Matrix44 *= None*) → Path

Returns a unit circle as a `Path` object, with the center at (0, 0, 0) and the radius of 1 drawing unit.

The arc spans from the start- to the end angle in counter-clockwise orientation. The end angle has to be greater than the start angle and the angle span has to be greater than 0.

Parameters

- **start_angle** – start angle in radians
- **end_angle** – end angle in radians (end_angle > start_angle!)
- **segments** – count of Bèzier-curve segments, default is one segment for each arc quarter (π/2)
- **transform** – transformation Matrix applied to the unit circle

ezdxf.path.**wedge**(*start_angle: float*, *end_angle: float*, *segments: int = 1*, *transform:* Matrix44 = *None*) → Path

Returns a wedge as a *Path* object, with the center at (0, 0, 0) and the radius of 1 drawing unit.

The arc spans from the start- to the end angle in counter-clockwise orientation. The end angle has to be greater than the start angle and the angle span has to be greater than 0.

Parameters

- **start_angle** – start angle in radians
- **end_angle** – end angle in radians (end_angle > start_angle!)
- **segments** – count of Bèzier-curve segments, default is one segment for each arc quarter (π/2)
- **transform** – transformation Matrix applied to the wedge

The *text2path* add-on provides additional functions to create paths from text strings and DXF text entities.

The Path Class

class ezdxf.path.**Path**

property end: *Vec3*

Path end point.

property has_curves: bool

Returns `True` if the path has any curve segments.

property has_lines: bool

Returns `True` if the path has any line segments.

property has_sub_paths: bool

Returns `True` if the path is a *Multi-Path* object which contains multiple sub-paths.

property is_closed: bool

Returns `True` if the start point is close to the end point.

property start: *Vec3*

Path start point, resetting the start point of an empty path is possible.

property user_data: Any

Attach arbitrary user data to a *Path* object. The user data is copied by reference, no deep copy is applied therefore a mutable state is shared between copies.

append_path(*path: Path*) → None

Append another path to this path. Adds a `self.line_to(path.start)` if the end of this path != the start of appended path.

approximate(*segments: int = 20*) → Iterator[*Vec3*]

Approximate path by vertices, *segments* is the count of approximation segments for each Bézier curve.

Does not yield any vertices for empty paths, where only a start point is present!

Approximation of *Multi-Path* objects is possible, but gaps are indistinguishable from line segments.

clockwise() → Path

Returns new *Path* in clockwise orientation.

Raises

TypeError – can't detect orientation of a *Multi-Path* object

clone() → Path

Returns a new copy of *Path* with shared immutable data.

close() → None

Close path by adding a line segment from the end point to the start point.

close_sub_path() → None

Close last sub-path by adding a line segment from the end point to the start point of the last sub-path. Behaves like *close()* for *Single-Path* instances.

control_vertices() → list[*ezdxf.math._vector.Vec3*]

Yields all path control vertices in consecutive order.

counter_clockwise() → Path

Returns new *Path* in counter-clockwise orientation.

Raises

TypeError – can't detect orientation of a *Multi-Path* object

curve3_to(*location:* UVec, *ctrl:* UVec) → None

Add a quadratic Bèzier-curve from actual path end point to *location*, *ctrl* is the control point for the quadratic Bèzier-curve.

curve4_to(*location:* UVec, *ctrl1:* UVec, *ctrl2:* UVec) → None

Add a cubic Bèzier-curve from actual path end point to *location*, *ctrl1* and *ctrl2* are the control points for the cubic Bèzier-curve.

extend_multi_path(*path: Path*) → None

Extend the path by another path. The source path is automatically a *Multi-Path* object, even if the previous end point matches the start point of the appended path. Ignores paths without any commands (empty paths).

flattening(*distance: float*, *segments: int = 16*) → Iterator[*Vec3*]

Approximate path by vertices and use adaptive recursive flattening to approximate Bèzier curves. The argument *segments* is the minimum count of approximation segments for each curve, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Does not yield any vertices for empty paths, where only a start point is present!

Flattening of *Multi-Path* objects is possible, but gaps are indistinguishable from line segments.

Parameters

- **distance** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count per Bézier curve

has_clockwise_orientation() → bool

Returns `True` if 2D path has clockwise orientation, ignores z-axis of all control vertices.

Raises

TypeError – can't detect orientation of a *Multi-Path* object

line_to(*location:* UVec) → None

Add a line from actual path end point to *location*.

move_to(*location:* UVec) → None

Start a new sub-path at *location*. This creates a gap between the current end-point and the start-point of the new sub-path. This converts the instance into a *Multi-Path* object.

If the *move_to()* command is the first command, the start point of the path will be reset to *location*.

reversed() → Path

Returns a new *Path* with reversed segments and control vertices.

sub_paths() → Iterator[Path]

Yield sub-path as *Single-Path* objects.

It is safe to call *sub_paths()* on any path-type: *Single-Path*, *Multi-Path* and *Empty-Path*.

transform(*m:* Matrix44) → Path

Returns a new transformed path.

Parameters
m – transformation matrix of type *Matrix44*

Disassemble

This module provide tools for the recursive decomposition of nested block reference structures into a flat stream of DXF entities and converting DXF entities into geometric primitives of *Path* and MeshBuilder objects encapsulated into intermediate *Primitive* classes.

Warning: Do not expect advanced vectorization capabilities: Text entities like TEXT, ATTRIB, ATTDEF and MTEXT get only a rough border box representation. The *text2path* add-on can convert text into paths. VIEWPORT, IMAGE and WIPEOUT are represented by their clipping path. Unsupported entities: all ACIS based entities, XREF, UNDERLAY, ACAD_TABLE, RAY, XLINE. Unsupported entities will be ignored.

Text Boundary Calculation

Text boundary calculations are based on monospaced (fixed-pitch, fixed-width, non-proportional) font metrics, which do not provide a good accuracy for text height calculation and much less accuracy for text width calculation. It is possible to improve this results by using the font support from the **optional** *Matplotlib* package.

Install Matplotlib from command line:

```
C:\> pip3 install matplotlib
```

The *Matplotlib* font support will improve the results for TEXT, ATTRIB and ATTDEF. The MTEXT entity has many advanced features which would require a full "Rich Text Format" rendering and that is far beyond the goals and capabilities of this library, therefore the boundary box for MTEXT will **never** be as accurate as in a dedicated CAD application.

Using the *Matplotlib* font support adds **runtime overhead**, therefore it is possible to deactivate the *Matplotlib* font support by setting the global option:

```
options.use_matplotlib_font_support = False
```

Flatten Complex DXF Entities

ezdxf.disassemble.**recursive_decompose**(*entities: Iterable[DXFEntity]*) → Iterable[DXFEntity]

Recursive decomposition of the given DXF entity collection into a flat stream of DXF entities. All block references (INSERT) and entities which provide a `virtual_entities()` method will be disassembled into simple DXF sub-entities, therefore the returned entity stream does not contain any INSERT entity.

Point entities will **not** be disassembled into DXF sub-entities, as defined by the current point style $PDMODE.

These entity types include sub-entities and will be decomposed into simple DXF entities:

- INSERT
- DIMENSION
- LEADER
- MLEADER
- MLINE

Decomposition of XREF, UNDERLAY and ACAD_TABLE entities is not supported.

Entity Deconstruction

These functions disassemble DXF entities into simple geometric objects like meshes, paths or vertices. The *Primitive* is a simplified intermediate class to use a common interface on various DXF entities.

ezdxf.disassemble.**make_primitive**(*entity: DXFEntity*, *max_flattening_distance=None*) → *Primitive*

Factory to create path/mesh primitives. The *max_flattening_distance* defines the max distance between the approximation line and the original curve. Use *max_flattening_distance* to override the default value.

Returns an **empty primitive** for unsupported entities. The *empty* state of a primitive can be checked by the property `is_empty`. The `path` and the `mesh` attributes of an empty primitive are `None` and the `vertices()` method yields no vertices.

ezdxf.disassemble.**to_primitives**(*entities: Iterable[DXFEntity]*, *max_flattening_distance: float | None = None*) → Iterable[*Primitive*]

Yields all DXF entities as path or mesh primitives. Yields unsupported entities as empty primitives, see *make_primitive()*.

Parameters

- **entities** – iterable of DXF entities
- **max_flattening_distance** – override the default value

ezdxf.disassemble.**to_meshes**(*primitives: Iterable[*Primitive*]*) → Iterable[MeshBuilder]

Yields all *MeshBuilder* objects from the given *primitives*. Ignores primitives without a defined mesh.

ezdxf.disassemble.**to_paths**(*primitives: Iterable[*Primitive*]*) → Iterable[Path]

Yields all *Path* objects from the given *primitives*. Ignores primitives without a defined path.

ezdxf.disassemble.**to_vertices**(*primitives: Iterable[*Primitive*]*) → Iterable[*Vec3*]

Yields all vertices from the given *primitives*. Paths will be flattened to create the associated vertices. See also *to_control_vertices()* to collect only the control vertices from the paths without flattening.

ezdxf.disassemble.**to_control_vertices**(*primitives: Iterable[*Primitive*]*) → Iterable[*Vec3*]

Yields all path control vertices and all mesh vertices from the given *primitives*. Like *to_vertices()*, but without flattening.

class ezdxf.disassemble.**Primitive**

Interface class for path/mesh primitives.

entity

Reference to the source DXF entity of this primitive.

max_flattening_distance

The *max_flattening_distance* attribute defines the max distance in drawing units between the approximation line and the original curve. Set the value by direct attribute access. (float) default = 0.01

property path: Path | None

Path representation or None, idiom to check if is a path representation (could be empty):

```
if primitive.path is not None:
    process(primitive.path)
```

property mesh: MeshBuilder | None

MeshBuilder representation or None, idiom to check if is a mesh representation (could be empty):

```
if primitive.mesh is not None:
    process(primitive.mesh)
```

property is_empty: bool

Returns *True* if represents an empty primitive which do not yield any vertices.

abstract vertices() → Iterable[*Vec3*]

Yields all vertices of the path/mesh representation as *Vec3* objects.

bbox(*fast=False*) → *BoundingBox*

Returns the *BoundingBox* of the path/mesh representation. Returns the precise bounding box for the path representation if *fast* is False, otherwise the bounding box for Bézier curves is based on their control points.

Bounding Box

The *ezdxf.bbox* module provide tools to calculate bounding boxes for many DXF entities, but not for all. The bounding box calculation is based on the *ezdxf.disassemble* module and therefore has the same limitation.

Warning: If accurate boundary boxes for text entities are important for you, read this first: *Text Boundary Calculation*. TL;DR: Boundary boxes for text entities are **not accurate!**

Unsupported DXF entities:

- All ACIS based types like BODY, 3DSOLID or REGION
- External references (XREF) and UNDERLAY object
- RAY and XRAY, extend into infinite
- ACAD_TABLE, no basic support - only preserved by *ezdxf*

Unsupported entities are silently ignored, filtering of these DXF types is not necessary.

The base type for bounding boxes is the *BoundingBox* class from the module *ezdxf.math*.

The *entities* iterable as input can be the whole modelspace, an entity query or any iterable container of DXF entities.

The Calculation of bounding boxes of curves is done by flattening the curve by a default flattening distance of 0.01. Set argument *flatten* to 0 to speedup the bounding box calculation by accepting less precision for curved objects by using only the control vertices.

The **optional** caching object *Cache* has to be instantiated by the user, this is only useful if the same entities will be processed multiple times.

Example usage with caching:

```
from ezdxf import bbox

msp = doc.modelspace()
cache = bbox.Cache()
# get overall bounding box
first_bbox = bbox.extents(msp, cache=cache)
# bounding box of all LINE entities
second_bbox = bbox.extend(msp.query("LINE"), cache=cache)
```

Functions

ezdxf.bbox.**extents**(*entities: Iterable[*DXFEntity*]*, *, *fast=False*, *cache:* Cache | *None = None*) → *BoundingBox*

Returns a single bounding box for all given *entities*.

If argument *fast* is `True` the calculation of Bézier curves is based on their control points, this may return a slightly larger bounding box. The *fast* mode also uses a simpler and mostly inaccurate text size calculation instead of the more precise but very slow calculation by *matplotlib*.

Hint: The fast mode is not much faster for non-text based entities, so using the slower default mode is not a big disadvantage if a more precise text size calculation is important.

---

ezdxf.bbox.**multi_flat**(*entities: Iterable[*DXFEntity*]*, *, *fast=False*, *cache:* Cache | *None = None*) → Iterable[*BoundingBox*]

Yields a bounding box for each of the given *entities*.

If argument *fast* is `True` the calculation of Bézier curves is based on their control points, this may return a slightly larger bounding box.

ezdxf.bbox.**multi_recursive**(*entities: Iterable[*DXFEntity*]*, *, *fast=False*, *cache:* Cache | *None = None*) → Iterable[*BoundingBox*]

Yields all bounding boxes for the given *entities* **or** all bounding boxes for their sub entities. If an entity (INSERT) has sub entities, only the bounding boxes of these sub entities will be yielded, **not** the bounding box of the entity (INSERT) itself.

If argument *fast* is `True` the calculation of Bézier curves is based on their control points, this may return a slightly larger bounding box.

Caching Strategies

Because *ezdxf* is not a CAD application, *ezdxf* does not manage data structures which are optimized for a usage by a CAD kernel. This means that the content of complex entities like block references or leaders has to be created on demand by DXF primitives on the fly. These temporarily created entities are called virtual entities and have no handle and are not stored in the entity database.

All this is required to calculate the bounding box of complex entities, and it is therefore a very time consuming task. By using a *Cache* object it is possible to speedup this calculations, but this is not a magically feature, it requires an understanding of what is happening under the hood to achieve any performance gains.

For a single bounding box calculation, without any reuse of entities it makes no sense of using a *Cache* object, e.g. calculation of the modelspace extents:

```
from pathlib import Path
import ezdxf
from ezdxf import bbox

CADKitSamples = Path(ezdxf.EZDXF_TEST_FILES) / 'CADKitSamples'

doc = ezdxf.readfile(CADKitSamples / 'A_000217.dxf')
cache = bbox.Cache()
ext = bbox.extents(doc.modelspace(), cache)

print(cache)
```

1226 cached objects and not a single cache hit:

```
Cache(n=1226, hits=0, misses=3273)
```

The result for using UUIDs to cache virtual entities is not better:

```
Cache(n=2206, hits=0, misses=3273)
```

Same count of hits and misses, but now the cache also references ~1000 virtual entities, which block your memory until the cache is deleted, luckily this is a small DXF file (~838 kB).

Bounding box calculations for multiple entity queries, which have overlapping entity results, using a *Cache* object may speedup the calculation:

```
doc = ezdxf.readfile(CADKitSamples / 'A_000217.dxf.dxf')
msp = doc.modelspace()
cache = bbox.Cache(uuid=False)

ext = bbox.extents(msp, cache)
print(cache)

# process modelspace again
ext = bbox.extents(msp, cache)
print(cache)
```

Processing the same data again leads some hits:

```
1st run: Cache(n=1226, hits=0, misses=3273)
2nd run: Cache(n=1226, hits=1224, misses=3309)
```

Using `uuid=True` leads not to more hits, but more cache entries:

```
1st run: Cache(n=2206, hits=0, misses=3273)
2nd run: Cache(n=2206, hits=1224, misses=3309)
```

Creating stable virtual entities by disassembling the entities at first leads to more hits:

```
from ezdxf import disassemble

entities = list(disassemble.recursive_decompose(msp))
cache = bbox.Cache(uuid=False)

bbox.extents(entities, cache)
print(cache)

bbox.extents(entities, cache)
print(cache)
```

First without UUID for stable virtual entities:

```
1st run: Cache(n=1037, hits=0, misses=4074)
2nd run: Cache(n=1037, hits=1037, misses=6078)
```

Using UUID for stable virtual entities leads to more hits:

```
1st run: Cache(n=2019, hits=0, misses=4074)
2nd run: Cache(n=2019, hits=2018, misses=4116)
```

But caching virtual entities needs also more memory.

In conclusion: Using a cache is only useful, if you often process **nearly the same data**; only then can an increase in performance be expected.

Cache Class

class ezdxf.bbox.**Cache**(*uuid=False*)

Caching object for *ezdxf.math.BoundingBox* objects.

Parameters
uuid – use UUIDs for virtual entities

has_data

`True` if the cache contains any bounding boxes

hits

misses

invalidate(*entities: Iterable[DXFEntity]*) → None

Invalidate cache entries for the given DXF *entities*.

If entities are changed by the user, it is possible to invalidate individual entities. Use with care - discarding the whole cache is the safer workflow.

Ignores entities which are not stored in cache.

Upright

The functions in this module can help to convert an inverted *OCS* defined by an extrusion vector (0, 0, -1) into a *WCS* aligned OCS defined by an extrusion vector (0, 0, 1).

This simplifies 2D entity processing for *ezdxf* users and creates DXF output for 3rd party DXF libraries which ignore the existence of the *OCS*.

Supported DXF entities:

- CIRCLE
- ARC
- ELLIPSE (WCS entity, flips only the extrusion vector)
- SOLID
- TRACE
- LWPOLYLINE
- POLYLINE (only 2D entities)
- HATCH
- MPOLYGON
- INSERT (block references)

Warning: The WCS representation of OCS entities with flipped extrusion vector is not 100% identical to the source entity, curve orientation and vertex order may change, see *additional explanation* below. A mirrored text represented by an extrusion vector (0, 0, -1) cannot represented by an extrusion vector (0, 0, 1), therefore this CANNOT work for text entities or entities including text: TEXT, ATTRIB, ATTDEF, MTEXT, DIMENSION, LEADER, MLEADER

Usage

The functions can be applied to any DXF entity without expecting errors or exceptions if the DXF entity is not supported or the extrusion vector differs from (0, 0, -1). This also means you can apply the functions multiple times to the same entities without any problems. A common case would be to upright all entities of the model space:

```
import ezdxf
from ezdxf.upright import upright_all

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
upright_all(msp)
# doing it again is no problem but also has no further effects
upright_all(msp)
```

Another use case is exploding block references (INSERT) which may include reflections (= scaling by negative factors) that can lead to inverted extrusion vectors.

```
for block_ref in msp.query("INSERT"):
    entities = block_ref.explode()  # -> EntityQuery object
    upright_all(entities)
```

Functions

ezdxf.upright.**upright**(*entity: DXFGraphic*) → None

Flips an inverted *OCS* defined by extrusion vector (0, 0, -1) into a *WCS* aligned *OCS* defined by extrusion vector (0, 0, 1). DXF entities with other extrusion vectors and unsupported DXF entities will be silently ignored. For more information about the limitations read the documentation of the *ezdxf.upright* module.

ezdxf.upright.**upright_all**(*entities: Iterable[DXFGraphic]*) → None

Call function *upright()* for all DXF entities in iterable *entities*:

```
upright_all(doc.modelspace())
```

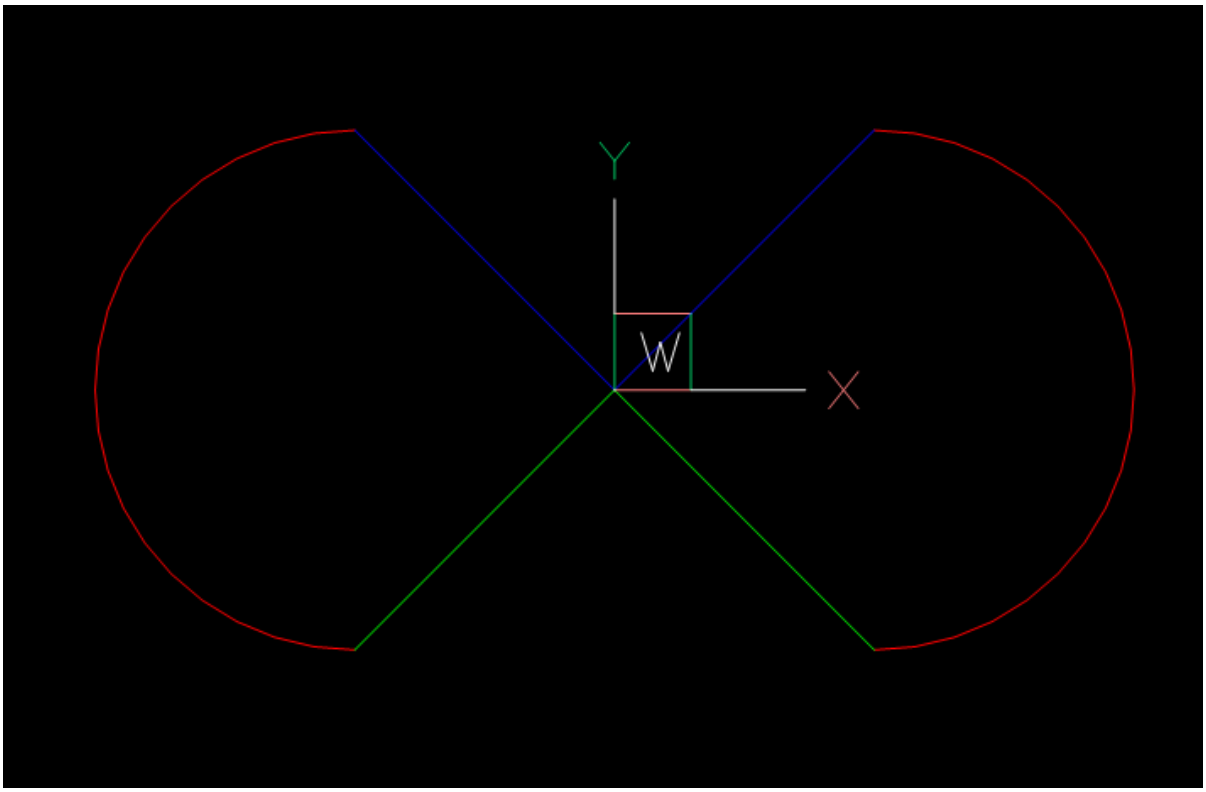
Additional Explanation

This example shows why the entities with an inverted OCS, extrusion vector is (0, 0, -1), are not exact the same as with an WCS aligned OCS, extrusion vector is (0, 0, 1).

Note: The ARC entity represents the curve **always** in counter-clockwise orientation around the extrusion vector.

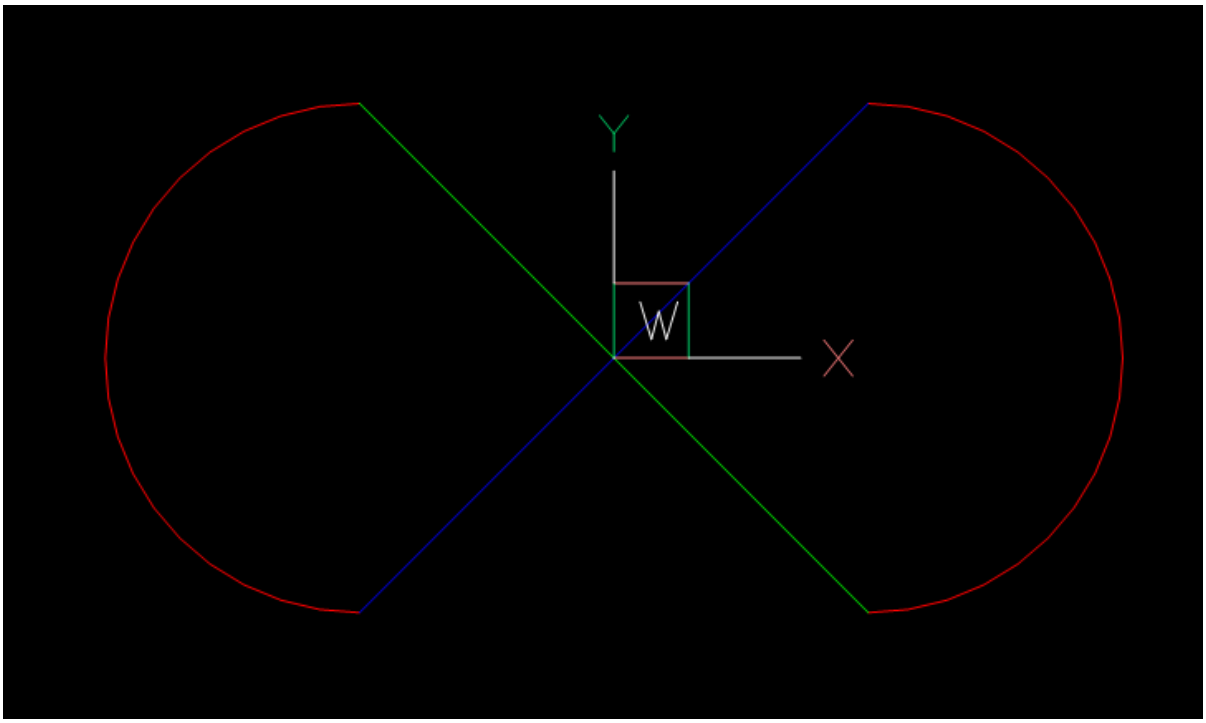
```
import ezdxf
from ezdxf.upright import upright
from ezdxf.math import Matrix44

doc = ezdxf.new()
msp = doc.modelspace()

arc = msp.add_arc(
    (5, 0),
    radius=5,
    start_angle=-90,
    end_angle=90,
    dxfattribs={"color": ezdxf.const.RED},
)
# draw lines to the start- and end point of the ARC
msp.add_line((0, 0), arc.start_point, dxfattribs={"color": ezdxf.const.GREEN})
msp.add_line((0, 0), arc.end_point, dxfattribs={"color": ezdxf.const.BLUE})

# copy arc
mirrored_arc = arc.copy()
msp.add_entity(mirrored_arc)

# mirror copy
mirrored_arc.transform(Matrix44.scale(-1, 1, 1))

# This creates an inverted extrusion vector:
assert mirrored_arc.dxf.extrusion.isclose((0, 0, -1))

# draw lines to the start- and end point of the mirrored ARC
msp.add_line((0, 0), mirrored_arc.start_point, dxfattribs={"color": ezdxf.const.GREEN}
↪)
msp.add_line((0, 0), mirrored_arc.end_point, dxfattribs={"color": ezdxf.const.BLUE})
```

Result without applying the *upright()* function - true mirroring:




```
...

# This creates an inverted extrusion vector:
assert mirrored_arc.dxf.extrusion.isclose((0, 0, -1))

start_point_inv = mirrored_arc.start_point
end_point_inv = mirrored_arc.end_point

upright(mirrored_arc)
# OCS is aligned with WCS:
assert mirrored_arc.dxf.extrusion.isclose((0, 0, 1))

# start- and end points are swapped after applying upright()
assert mirrored_arc.start_point.isclose(end_point_inv)
assert mirrored_arc.end_point.isclose(start_point_inv)

# draw lines to the start- and end point of the mirrored ARC
msp.add_line((0, 0), mirrored_arc.start_point, dxfattribs={"color": ezdxf.const.GREEN}
↪)
msp.add_line((0, 0), mirrored_arc.end_point, dxfattribs={"color": ezdxf.const.BLUE})
```

Result after applying the *upright()* function - false mirroring:



To avoid this issue the ARC entity would have to represent the curve in clockwise orientation around the extrusion vector (0, 0, 1), which is not possible!

Note: The shape of the mirrored arcs is the same for both extrusion vectors, but the start- and the end points are swapped (reversed vertex order)!

Reorder

Tools to reorder DXF entities by handle or a special sort handle mapping.

Such reorder mappings are stored only in layouts as `Modelspace`, `Paperspace` or `BlockLayout`, and can be retrieved by the method `get_redraw_order()`.

Each entry in the handle mapping replaces the actual entity handle, where the "0" handle has a special meaning, this handle always shows up at last in ascending ordering.

`ezdxf.reorder.`**`ascending`**(*entities: Iterable[*DXFGraphic*]*, *mapping: dict | Iterable[tuple[str, str]] | None = None*) → Iterable[*DXFGraphic*]

Yields entities in ascending handle order.

The sort-handle doesn't have to be the entity handle, every entity handle in *mapping* will be replaced by the given sort-handle, *mapping* is an iterable of 2-tuples (entity_handle, sort_handle) or a dict (entity_handle, sort_handle). Entities with equal sort-handles show up in source entities order.

Parameters

- **`entities`** – iterable of `DXFGraphic` objects
- **`mapping`** – iterable of 2-tuples (entity_handle, sort_handle) or a handle mapping as dict.

`ezdxf.reorder.`**`descending`**(*entities: Iterable[*DXFGraphic*]*, *mapping: dict | Iterable[tuple[str, str]] | None = None*) → Iterable[*DXFGraphic*]

Yields entities in descending handle order.

The sort-handle doesn't have to be the entity handle, every entity handle in *mapping* will be replaced by the given sort-handle, *mapping* is an iterable of 2-tuples (entity_handle, sort_handle) or a dict (entity_handle, sort_handle). Entities with equal sort-handles show up in reversed source entities order.

Parameters

- **entities** – iterable of `DXFGraphic` objects
- **mapping** – iterable of 2-tuples (entity_handle, sort_handle) or a handle mapping as dict.

Transform

New in version 1.1.

This module provides functions to apply transformations to multiple DXF entities inplace in a more convenient and safe way:

```
import math

import ezdxf
from ezdxf import transform

doc = ezdxf.readfile("my.dxf")
msp = doc.modelspace()

log = transform.matrix(msp, m=transform.Matrix44.rotate_z(math.pi/2))

# or more simple
log = transform.z_rotate(msp, math.pi/2)
```

All functions handle errors by collecting them in an logging object without raising an error. The input *entities* are an iterable of *DXFEntity*, which can be any layout, *EntityQuery* or just a list/sequence of entities and virtual entities are supported as well.

matrix	Transforms the given *entities* inplace by the transformation matrix *m*, non-uniform scaling is not supported.
matrix_ext	Transforms the given *entities* inplace by the transformation matrix *m*, non-uniform scaling is supported.
translate	Translates (moves) *entities* inplace by the *offset* vector.
scale_uniform	Scales *entities* inplace by a *factor* in all axis.
scale	Scales *entities* inplace by the factors *sx* in x-axis, *sy* in y-axis and *sz* in z-axis.
x_rotate	Rotates *entities* inplace by *angle* in radians about the x-axis.
y_rotate	Rotates *entities* inplace by *angle* in radians about the y-axis.
z_rotate	Rotates *entities* inplace by *angle* in radians about the x-axis.
axis_rotate	Rotates *entities* inplace by *angle* in radians about the rotation axis starting at the origin pointing in *axis* direction.

ezdxf.transform.**matrix**(*entities: Iterable[DXFEntity]*, *m:* Matrix44) → *Logger*

Transforms the given *entities* inplace by the transformation matrix *m*, non-uniform scaling is not supported. The function logs errors and does not raise errors for unsupported entities or transformations that cannot be performed, see enum *Error*. The *matrix()* function supports virtual entities as well.

ezdxf.transform.**matrix_ext**(*entities: Iterable[DXFEntity]*, *m:* Matrix44) → *Logger*

Transforms the given *entities* inplace by the transformation matrix *m*, non-uniform scaling is supported. The function converts circular arcs into ellipses to perform non-uniform scaling. The function logs errors and does not raise errors for unsupported entities or transformation errors, see enum *Error*.

Important: The *matrix_ext()* function does not support type conversion for virtual entities e.g. non-uniform scaling for CIRCLE, ARC or POLYLINE with bulges.

---

ezdxf.transform.**translate**(*entities: Iterable[DXFEntity]*, *offset:* UVec) → *Logger*

Translates (moves) *entities* inplace by the *offset* vector.

ezdxf.transform.**scale_uniform**(*entities: Iterable[DXFEntity]*, *factor: float*) → *Logger*

Scales *entities* inplace by a *factor* in all axis. Scaling factors smaller than *MIN_SCALING_FACTOR* are ignored.

ezdxf.transform.**scale**(*entities: Iterable[DXFEntity]*, *sx: float*, *sy: float*, *sz: float*) → *Logger*

Scales *entities* inplace by the factors *sx* in x-axis, *sy* in y-axis and *sz* in z-axis. Scaling factors smaller than *MIN_SCALING_FACTOR* are ignored.

Important: The *scale()* function does not support virtual entities!

---

ezdxf.transform.**x_rotate**(*entities: Iterable[DXFEntity]*, *angle: float*) → *Logger*

Rotates *entities* inplace by *angle* in radians about the x-axis.

ezdxf.transform.**y_rotate**(*entities: Iterable[DXFEntity]*, *angle: float*) → *Logger*

Rotates *entities* inplace by *angle* in radians about the y-axis.

ezdxf.transform.**z_rotate**(*entities: Iterable[DXFEntity]*, *angle: float*) → *Logger*

Rotates *entities* inplace by *angle* in radians about the x-axis.

ezdxf.transform.**axis_rotate**(*entities: Iterable[DXFEntity]*, *axis:* UVec, *angle: float*) → *Logger*

Rotates *entities* inplace by *angle* in radians about the rotation axis starting at the origin pointing in *axis* direction.

ezdxf.transform.**MIN_SCALING_FACTOR**

Minimal scaling factor: 1e-12

class ezdxf.transform.**Error**

TRANSFORMATION_NOT_SUPPORTED

Entities without transformation support.

NON_UNIFORM_SCALING_ERROR

Circular arcs (CIRCLE, ARC, bulges in POLYLINE and LWPOLYLINE entities) cannot be scaled non-uniformly.

INSERT_TRANSFORMATION_ERROR

INSERT entities cannot represent a non-orthogonal target coordinate system. Maybe exploding the INSERT entities (recursively) beforehand can solve this issue, see function *ezdxf.disassemble.recursive_decompose()*.

VIRTUAL_ENTITY_NOT_SUPPORTED

Transformation not supported for virtual entities e.g. non-uniform scaling for CIRCLE, ARC or POLYLINE with bulges

class ezdxf.transform.**Logger**

A Sequence of errors as *Logger.Entry* instances.

class Entry

Named tuple representing a logger entry.

error

Error enum

msg

error message as string

entity

DXF entity which causes the error

__len__() → int

Returns the count of error entries.

__getitem__(*index: int*) → *Entry*

Returns the error entry at *index*.

__iter__() → Iterator[*Entry*]

Iterates over all error entries.

messages() → list[str]

Returns all error messages as list of strings.

Math Construction Tools

These are links to tools in the *ezdxf.math* core module:

ezdxf.math.ConstructionRay	Construction tool for infinite 2D rays.
ezdxf.math.ConstructionLine	Construction tool for 2D lines.
ezdxf.math.ConstructionCircle	Construction tool for 2D circles.
ezdxf.math.ConstructionArc	Construction tool for 2D arcs.
ezdxf.math.ConstructionEllipse	Construction tool for 3D ellipsis.
ezdxf.math.ConstructionBox	Construction tool for 2D rectangles.
ezdxf.math.ConstructionPolyline	Construction tool for 3D polylines.
ezdxf.math.Shape2d	Construction tools for 2D shapes.
ezdxf.math.BSpline	B-spline construction tool.
ezdxf.math.Bezier4P	Implements an optimized cubic Bézier curve for exact 4 control points.
ezdxf.math.Bezier3P	Implements an optimized quadratic Bézier curve for exact 3 control points.
ezdxf.math.Bezier	Generic Bézier curve of any degree.
ezdxf.math.BezierSurface	BezierSurface defines a mesh of *m* x *n* control points.
ezdxf.math.EulerSpiral	This class represents an euler spiral (clothoid) for *curvature* (Radius of curvature).

6.9.9 Custom Data

Custom XDATA

The classes *XDataUserList* and *XDataUserDict* manage custom user data stored in the XDATA section of a DXF entity. For more information about XDATA see reference section: *Extended Data (XDATA)*

These classes store only a limited set of data types with fixed group codes and the types are checked by `isinstance()` so a `Vec3` object can not be replaced by a (x, y, z)-tuple:

Group Code	Data Type
1000	str, limited to 255 characters, line breaks `"\n"` and `"\r"` are not allowed
1010	Vec3
1040	float
1071	32-bit int, restricted by the DXF standard not by Python!

Strings are limited to 255 characters, line breaks `"\n"` and `"\r"` are not allowed.

This classes assume a certain XDATA structure and therefore can not manage arbitrary XDATA!

This classes do not create the required AppID table entry, only the default AppID "EZDXF" exist by default. Setup a new AppID in the AppID table: `doc.appids.add("MYAPP")`.

For usage look at this example at github or go to the tutorial: *Storing Custom Data in DXF Files*.

See also:

- Tutorial: *Storing Custom Data in DXF Files*
- Example at github
- XDATA reference: *Extended Data (XDATA)*
- XDATA management class: *XData*

XDataUserList

class ezdxf.entities.xdata.**XDataUserList**

Manage user data as a named list-like object in XDATA. Multiple user lists with different names can be stored in a single *XData* instance for a single AppID.

Recommended usage by context manager *entity()*:

```
with XDataUserList.entity(entity, name="MyList", appid="MYAPP") as ul:
    ul.append("The value of PI")  # str "\n" and "\r" are not allowed
    ul.append(3.141592)  # float
    ul.append(1) # int
    ul.append(Vec3(1, 2, 3)) # Vec3

    # invalid data type raises DXFTypeError
    ul.append((1, 2, 3))  # tuple instead of Vec3

    # retrieve a single value
    s = ul[0]

    # store whole content into a Python list
    data = list(ul)
```

Implements the `MutableSequence` interface.

xdata

The underlying *XData* instance.

__init__ (*xdata:* XData | *None* = *None*, *name='DefaultList'*, *appid='EZDXF'*)

Setup a XDATA user list *name* for the given *appid*.

The data is stored in the given *xdata* object, or in a new created *XData* instance if `None`. Changes of the content has to be committed at the end to be stored in the underlying *xdata* object.

Parameters

- **xdata** (*XData*) – underlying *XData* instance, if `None` a new one will be created
- **name** (*str*) – name of the user list
- **appid** (*str*) – application specific AppID

__str__ ()

Return str(self).

__len__ () → int

Returns len(self).

__getitem__ (*item*)

Get self[item].

__setitem__ (*item*, *value*)

Set self[item] to value.

__delitem__ (*item*)

Delete self[item].

classmethod entity (*entity:* DXFEntity, *name='DefaultList'*, *appid='EZDXF'*) → Iterator[*XDataUserList*]

Context manager to manage a XDATA list *name* for a given DXF *entity*. Appends the user list to the existing *XData* instance or creates new *XData* instance.

Parameters

- **entity** (*DXFEntity*) – target DXF entity for the XDATA
- **name** (*str*) – name of the user list
- **appid** (*str*) – application specific AppID

commit () → None

Store all changes to the underlying *XData* instance. This call is not required if using the *entity()* context manager.

Raises

- ***DXFValueError*** – invalid chars `"\n"` or `"\r"` in a string
- ***DXFTypeError*** – invalid data type

XDataUserDict

class ezdxf.entities.xdata.**XDataUserDict**

Manage user data as a named dict-like object in XDATA. Multiple user dicts with different names can be stored in a single *XData* instance for a single AppID. The keys have to be strings.

Recommended usage by context manager *entity()*:

```
with XDataUserDict.entity(entity, name="MyDict", appid="MYAPP") as ud:
    ud["comment"] = "The value of PI"  # str "\n" and "\r" are not allowed
    ud["pi"] = 3.141592  # float
    ud["number"] = 1 # int
    ud["vertex"] = Vec3(1, 2, 3) # Vec3

    # invalid data type raises DXFTypeError
    ud["vertex"] = (1, 2, 3)  # tuple instead of Vec3

    # retrieve single values
    s = ud["comment"]
    pi = ud.get("pi", 3.141592)

    # store whole content into a Python dict
    data = dict(ud)
```

Implements the `MutableMapping` interface.

The data is stored in XDATA like a *XDataUserList* by (key, value) pairs, therefore a *XDataUserDict* can also be loaded as *XDataUserList*. It is not possible to distinguish a *XDataUserDict* from a *XDataUserList* except by the name of the data structure.

xdata

The underlying *XData* instance.

__init__(*xdata:* XData | *None* = *None*, *name='DefaultDict'*, *appid='EZDXF'*)

Setup a XDATA user dict *name* for the given *appid*.

The data is stored in the given *xdata* object, or in a new created *XData* instance if `None`. Changes of the content has to be committed at the end to be stored in the underlying *xdata* object.

Parameters

- **xdata** (`XData`) – underlying *XData* instance, if `None` a new one will be created
- **name** (*str*) – name of the user list
- **appid** (*str*) – application specific AppID

__str__()

Return str(self).

__len__()

Returns len(self).

__getitem__(*key*)

Get self[key].

__setitem__(*key*, *item*)

Set self[key] to value, key has to be a string.

Raises

DXFTypeError – key is not a string

__delitem__(*key*)

Delete self[key].

discard(*key*)

Delete self[key], without raising a `KeyError` if *key* does not exist.

__iter__()

Implement iter(self).

classmethod entity(*entity:* DXFEntity, *name='DefaultDict'*, *appid='EZDXF'*) → Iterator[*XDataUserDict*]

Context manager to manage a XDATA dict *name* for a given DXF *entity*. Appends the user dict to the existing *XData* instance or creates new *XData* instance.

Parameters

- **entity** (*DXFEntity*) – target DXF entity for the XDATA
- **name** (*str*) – name of the user list
- **appid** (*str*) – application specific AppID

commit() → None

Store all changes to the underlying *XData* instance. This call is not required if using the *entity()* context manager.

Raises

- ***DXFValueError*** – invalid chars `"\n"` or `"\r"` in a string
- ***DXFTypeError*** – invalid data type

Custom XRecord

The *UserRecord* and *BinaryRecord* classes help to store custom data in DXF files in *XRecord* objects a simple and safe way. This way requires DXF version R2000 or later, for DXF version R12 the only way to store custom data is *Extended Data (XDATA)*.

The *UserRecord* stores Python types and nested container types: `int`, `float`, `str`, *Vec2*, *Vec3*, `list` and `dict`.

Requirements for Python structures:

- The top level structure has to be a `list`.
- Strings has to have max. 2049 characters and can not contain line breaks `"\\n"` or `"\\r"`.
- Dict keys have to be simple Python types: `int`, `float`, `str`.

DXF Tag layout for Python types and structures stored in the *XRecord* object:

Only for the *UserRecord* the first tag is (2, user record name).

Type	DXF Tag(s)
str	(1, value) string with less than 2050 chars and including no line breaks
int	(90, value) int 32-bit, restricted by the DXF standard not by Python!
float	(40, value) “C” double
Vec2	(10, x), (20, y)
Vec3	(10, x) (20, y) (30, z)
list	starts with (2, “[”) and ends with (2, “]”)
dict	starts with (2, “{”) and ends with (2, “}”)

The *BinaryRecord* stores arbitrary binary data as BLOB.

Storage size limits of XRECORD according the DXF reference:

> "This object is similar in concept to XDATA but is not limited by size or order."

For usage look at this example at github or go to the tutorial: *Storing Custom Data in DXF Files*.

See also:

- Tutorial: *Storing Custom Data in DXF Files*
- Example at github
- *ezdxf.entities.XRecord*

UserRecord

class ezdxf.urecord.**UserRecord**

xrecord

The underlying *XRecord* instance

name

The name of the *UserRecord*, an arbitrary string with less than 2050 chars and including no line breaks.

data

The Python data. The top level structure has to be a list (`MutableSequence`). Inside this container the following Python types are supported: str, int, float, Vec2, Vec3, list, dict

Nested data structures are supported list or/and dict in list or dict. Dict keys have to be simple Python types: int, float, str.

property handle: str | None

DXF handle of the underlying *XRecord* instance.

__init__ (*xrecord:* XRecord | *None* = *None*, *, *name: str* = *DEFAULT_NAME*, *doc:* Drawing | *None* = *None*)

Setup a *UserRecord* with the given *name*.

The data is stored in the given *xrecord* object, or in a new created *XRecord* instance if `None`. If *doc* is not `None` the new xrecord is added to the OBJECTS section of the DXF document.

Changes of the content has to be committed at the end to be stored in the underlying *xrecord* object.

Parameters

- **xrecord** (*XRecord*) – underlying *XRecord* instance, if `None` a new one will be created
- **name** (*str*) – name of the user list
- **doc** (*Drawing*) – DXF document or `None`

__str__ ()

Return str(self).

commit () → XRecord

Store *data* in the underlying *XRecord* instance. This call is not required if using the class by the `with` statement.

Raises

- ***DXFValueError*** – invalid chars `"\n"` or `"\r"` in a string

- ***DXFTypeError*** – invalid data type

BinaryRecord

class ezdxf.urecord.**BinaryRecord**

xrecord

The underlying *XRecord* instance

data

The binary data as bytes, bytearray or memoryview.

property handle: str | None

DXF handle of the underlying *XRecord* instance.

__init__ (*xrecord:* XRecord | *None* = *None*, *, *doc:* Drawing | *None* = *None*)

Setup a *BinaryRecord*.

The data is stored in the given *xrecord* object, or in a new created *XRecord* instance if `None`. If *doc* is not `None` the new xrecord is added to the OBJECTS section of the DXF document.

Changes of the content has to be committed at the end to be stored in the underlying *xrecord* object.

Parameters

- **xrecord** (`XRecord`) – underlying *XRecord* instance, if `None` a new one will be created
- **doc** (`Drawing`) – DXF document or `None`

__str__ () → str

Return str(self).

commit () → XRecord

Store binary *data* in the underlying *XRecord* instance. This call is not required if using the class by the `with` statement.

6.9.10 Tools

Functions

DXF Unicode Decoder

The DXF format uses a special form of unicode encoding: "\U+xxxx".

To avoid a speed penalty such encoded characters are not decoded automatically by the regular loading function:func:*ezdxf.readfile*, only the *recover* module does the decoding automatically, because this loading mode is already slow.

This kind of encoding is most likely used only in older DXF versions, because since DXF R2007 the whole DXF file is encoded in `utf8` and a special unicode encoding is not necessary.

The *ezdxf.has_dxf_unicode()* and *ezdxf.decode_dxf_unicode()* are new support functions to decode unicode characters "\U+xxxx" manually.

ezdxf.**has_dxf_unicode**(*s: str*) → bool

Detect if string *s* contains encoded DXF unicode characters "\U+xxxx".

ezdxf.**decode_dxf_unicode**(*s: str*) → str

Decode DXF unicode characters "\U+xxxx" in string *s*.

Tools

Some handy tool functions used internally by ezdxf.

ezdxf.tools.**juliandate**(*date: datetime*) → float

ezdxf.tools.**calendardate**(*juliandate: float*) → datetime

ezdxf.tools.**set_flag_state**(*flags: int*, *flag: int*, *state: bool = True*) → int

Set/clear binary *flag* in data *flags*.

Parameters

- **flags** – data value
- **flag** – flag to set/clear
- **state** – True for setting, False for clearing

ezdxf.tools.**guid**() → str

Returns a general unique ID, based on uuid.uuid4().

This function creates a GUID for the header variables $VERSIONGUID and $FINGERPRINTGUID, which matches the AutoCAD pattern {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}.

ezdxf.tools.**bytes_to_hexstr**(*data: bytes*) → str

Returns *data* bytes as plain hex string.

ezdxf.tools.**suppress_zeros**(*s: str*, *leading: bool = False*, *trailing: bool = True*)

Suppress trailing and/or leading 0 of string *s*.

Parameters

- **s** – data string
- **leading** – suppress leading 0
- **trailing** – suppress trailing 0

ezdxf.tools.**normalize_text_angle**(*angle: float*, *fix_upside_down=True*) → float

Normalizes text *angle* to the range from 0 to 360 degrees and fixes upside down text angles.

Parameters

- **angle** – text angle in degrees
- **fix_upside_down** – rotate upside down text angle about 180 degree

SAT Format "Encryption"

ezdxf.tools.crypt.**encode**(*text_lines: Iterable[str]*) → Iterable[str]

Encode the Standard *ACIS* Text (SAT) format by AutoCAD "encryption" algorithm.

ezdxf.tools.crypt.**decode**(*text_lines: Iterable[str]*) → Iterable[str]

Decode the Standard *ACIS* Text (SAT) format "encrypted" by AutoCAD.

GfxAttribs

The *ezdxf.gfxattribs* module provides the *GfxAttribs* class to create valid attribute dictionaries for the most often used DXF attributes supported by all graphical DXF entities. The advantage of using this class is auto-completion support by IDEs and an instant validation of the attribute values.

```
import ezdxf
from ezdxf.gfxattribs import GfxAttribs

doc = ezdxf.new()
msp = doc.modelspace()

attribs = GfxAttribs(layer="MyLayer", color=ezdxf.colors.RED)
line = msp.add_line((0, 0), (1, 0), dxfattribs=attribs)
circle = msp.add_circle((0, 0), radius=1.0, dxfattribs=attribs)

# Update DXF attributes of existing entities:
attribs = GfxAttribs(layer="MyLayer2", color=ezdxf.colors.BLUE)

# Convert GfxAttribs() to dict(), but this method cannot reset
# attributes to the default values like setting layer to "0".
line.update_dxf_attribs(dict(attribs))

# Using GfxAttribs.asdict(default_values=True), can reset attributes to the
# default values like setting layer to "0", except for true_color and
# transparency, which do not have default values, their absence is the
# default value.
circle.update_dxf_attribs(attribs.asdict(default_values=True))

# Remove true_color and transparency by assigning None
attribs.transparency = None  # reset to transparency by layer!
attribs.rgb = None
```

Validation features:

- **layer** - string which can not contain certain characters: <>/\":;?*=`
- **color** - *AutoCAD Color Index (ACI)* value as integer in the range from 0 to 257
- **rgb** - true color value as (red, green, blue) tuple, all channel values as integer values in the range from 0 to 255
- **linetype** - string which can not contain certain characters: <>/\":;?*=`, does not check if the linetype exists
- **lineweight** - integer value in the range from 0 to 211, see *Lineweights* for valid values
- **transparency** - float value in the range from 0.0 to 1.0 and -1.0 for transparency by block
- **ltscale** - float value > 0.0

class ezdxf.gfxattribs.**GfxAttribs**(*, *layer: str = DEFAULT_LAYER*, *color: int = DEFAULT_ACI_COLOR*, *rgb: Tuple[int, int, int] | None = None*, *linetype: str = DEFAULT_LINETYPE*, *lineweight: int = DEFAULT_LINEWEIGHT*, *transparency: float | None = None*, *ltscale: float = DEFAULT_LTSCALE*)

Represents often used DXF attributes of graphical entities.

Parameters

- **layer** (*str*) – layer name as string
- **color** (*int*) – *AutoCAD Color Index (ACI)* color value as integer
- **rgb** – RGB true color (red, green, blue) tuple, each channel value in the range from 0 to 255, `None` for not set
- **linetype** (*str*) – linetype name, does not check if the linetype exist!
- **lineweight** (*int*) – see *Lineweights* documentation for valid values
- **transparency** (*float*) – transparency value in the range from 0.0 to 1.0, where 0.0 is opaque and 1.0 if fully transparent, -1.0 for transparency by block, `None` for transparency by layer
- **ltscale** (*float*) – linetype scaling factor > 0.0, default factor is 1.0

Raises

DXFValueError – invalid attribute value

property layer: str

layer name

property color: int

AutoCAD Color Index (ACI) color value

property rgb: Tuple[int, int, int] | None

true color value as (red, green, blue) tuple, `None` for not set

property linetype: str

linetype name

property lineweight: int

property transparency: float | None

transparency value from 0.0 for opaque to 1.0 is fully transparent, -1.0 is for transparency by block and `None` if for transparency by layer

property ltscale: float

linetype scaling factor

__str__() → str

Return str(self).

__repr__() → str

Return repr(self).

__iter__() → Iterator[tuple[str, Any]]

Returns iter(self).

asdict(*default_values=False*) → dict[str, Any]

Returns the DXF attributes as `dict`, returns also the default values if argument *default_values* is `True`. The true_color and transparency attributes do not have default values, the absence of these attributes is the default value.

items(*default_values=False*) → list[tuple[str, Any]]

Returns the DXF attributes as list of name, value pairs, returns also the default values if argument *default_values* is `True`. The true_color and transparency attributes do not have default values, the absence of these attributes is the default value.

classmethod load_from_header(*doc:* Drawing) → *GfxAttribs*

Load default DXF attributes from the HEADER section.

There is no default true color value and the default transparency is not stored in the HEADER section.

Loads following header variables:

- `$CLAYER` - current layer name
- `$CECOLOR` - current ACI color
- `$CELTYPE` - current linetype name
- `$CELWEIGHT` - current lineweight
- `$CELTSCALE` - current linetype scaling factor

write_to_header(*doc:* Drawing) → None

Write DXF attributes as default values to the HEADER section.

Writes following header variables:

- `$CLAYER` - current layer name, if a layer table entry exist in *doc*
- `$CECOLOR` - current ACI color
- `$CELTYPE` - current linetype name, if a linetype table entry exist in *doc*
- `$CELWEIGHT` - current lineweight
- `$CELTSCALE` - current linetype scaling factor

classmethod from_entity(*entity:* DXFEntity) → *GfxAttribs*

Get the graphical attributes of an *entity* as `GfxAttribs` object.

classmethod from_dict(*d: dict[str, Any]*) → *GfxAttribs*

Construct `GfxAttribs` from a dictionary of raw DXF values.

Supported attributes are:

- layer: layer name as string
- color: *AutoCAD Color Index (ACI)* value as int
- true_color: raw DXF integer value for RGB colors
- rgb: RGB tuple of int or `None`
- linetype: linetype name as string
- lineweight: lineweight as int, see basic concept of *Lineweights*
- transparency: raw DXF integer value of transparency or a float in the range from 0.0 to 1.0
- ltscale: linetype scaling factor as float

Text Tools

MTextEditor

class ezdxf.tools.text.**MTextEditor**(*text: str = ''*)

The *MTextEditor* is a helper class to build MTEXT content strings with support for inline codes to change color, font or paragraph properties. The result is always accessible by the *text* attribute or the magic *__str__()* function as `str(MTextEditor("text"))`.

All text building methods return *self* to implement a floating interface:

```
e = MTextEditor("This example ").color("red").append("switches color to red.")
mtext = msp.add_mtext(str(e))
```

The initial text height, color, text style and so on is determined by the DXF attributes of the *MText* entity.

Warning: The *MTextEditor* assembles just the inline code, which has to be parsed and rendered by the target CAD application, *ezdxf* has no influence to that result.

Keep inline formatting as simple as possible, don't test the limits of its capabilities, this will not work across different CAD applications and keep the formatting in a logic manner like, do not change paragraph properties in the middle of a paragraph.

There is no official documentation for the inline codes!

Parameters
text – init value of the MTEXT content string.

text

The MTEXT content as a simple string.

append(*text: str*) → *MTextEditor*

Append *text*.

__iadd__(*text: str*) → *MTextEditor*

Append *text*:

```
e = MTextEditor("First paragraph.\P")
e += "Second paragraph.\P")
```

__str__() → str

Returns the MTEXT content attribute *text*.

clear()

Reset the content to an empty string.

font(*name: str*, *bold: bool = False*, *italic: bool = False*) → *MTextEditor*

Set the text font by the font family name. Changing the font height should be done by the *height()* or the *scale_height()* method. The font family name is the name shown in font selection widgets in desktop applications: "Arial", "Times New Roman", "Comic Sans MS". Switching the codepage is not supported.

Parameters

- **name** – font family name
- **bold** – flag

- **italic** – flag

height (*height: float*) → *MTextEditor*

Set the absolute text height in drawing units.

scale_height (*factor: float*) → *MTextEditor*

Scale the text height by a *factor*. This scaling will accumulate, which means starting at height 2.5 and scaling by 2 and again by 3 will set the text height to 2.5 x 2 x 3 = 15. The current text height is not stored in the *MTextEditor*, you have to track the text height by yourself! The initial text height is stored in the *MText* entity as DXF attribute *char_height*.

width_factor (*factor: float*) → *MTextEditor*

Set the absolute text width factor.

char_tracking_factor (*factor: float*) → *MTextEditor*

Set the absolute character tracking factor.

oblique (*angle: int*) → *MTextEditor*

Set the text oblique angle in degrees, vertical is 0, a value of 15 will lean the text 15 degree to the right.

color (*name: str*) → *MTextEditor*

Set the text color by color name: “red”, “yellow”, “green”, “cyan”, “blue”, “magenta” or “white”.

aci (*aci: int*) → *MTextEditor*

Set the text color by *AutoCAD Color Index (ACI)* in range [0, 256].

rgb (*rgb: Tuple[int, int, int]*) → *MTextEditor*

Set the text color as RGB value.

underline (*text: str*) → *MTextEditor*

Append *text* with a line below the text.

overline (*text: str*) → *MTextEditor*

Append *text* with a line above the text.

strike_through (*text: str*) → *MTextEditor*

Append *text* with a line through the text.

group (*text: str*) → *MTextEditor*

Group *text*, all properties changed inside a group are reverted at the end of the group. AutoCAD supports grouping up to 8 levels.

stack (*upr: str*, *lwr: str*, *t: str = '^'*) → *MTextEditor*

Append stacked text *upr* over *lwr*, argument *t* defines the kind of stacking, the space “ “ after the “^” will be added automatically to avoid caret decoding:

```
"^": vertical stacked without divider line, e.g. \SA^ B:
     A
     B

"/": vertical stacked with divider line,  e.g. \SX/Y:
     X
     -
     Y

"#": diagonal stacked, with slanting divider line, e.g. \S1#4:
     1/4
```

paragraph (*props:* ParagraphProperties) → *MTextEditor*

Set paragraph properties by a `ParagraphProperties` object.

bullet_list (*indent: float*, *bullets: Iterable[str]*, *content: Iterable[str]*) → *MTextEditor*

Build bulleted lists by utilizing paragraph indentation and a tabulator stop. Any string can be used as bullet. Indentation is a multiple of the initial MTEXT char height (see also docs about `ParagraphProperties`), which means indentation in drawing units is `MText.dxf.char_height` x *indent*.

Useful UTF bullets:

- "bull" U+2022 = • (Alt Numpad 7)
- "circle" U+25CB = ○ (Alt Numpad 9)

For numbered lists just use numbers as bullets:

```
MTextEditor.bullet_list(
    indent=2,
    bullets=["1.", "2."],
    content=["first", "second"],
)
```

Parameters

- **indent** – content indentation as multiple of the initial MTEXT char height
- **bullets** – iterable of bullet strings, e.g. `["-"] * 3`, for 3 dashes as bullet strings
- **content** – iterable of list item strings, one string per list item, list items should not contain new line or new paragraph commands.

Constants stored in the `MTextEditor` class:

NEW_LINE	`'\P'`
NEW_PARAGRAPH	`'\P'`
NEW_COLUMN	`'\N`
UNDERLINE_START	`'\L'`
UNDERLINE_STOP	`'\l'`
OVERSTRIKE_START	`'\O'`
OVERSTRIKE_STOP	`'\o'`
STRIKE_START	`'\K'`
STRIKE_STOP	`'\k'`
ALIGN_BOTTOM	`'\A0;'`
ALIGN_MIDDLE	`'\A1;'`
ALIGN_TOP	`'\A2;'`
NBSP	`'\~'`
TAB	`'^I'`

class ezdxf.tools.text.**ParagraphProperties** (*indent=0*, *left=0*, *right=0*, *align=DEFAULT*, *tab_stops=[]*)

Stores all known MTEXT paragraph properties in a `NamedTuple`. Indentations and tab stops are multiples of the default text height `MText.dxf.char_height`. E.g. `char_height` is 0.25 and `indent` is 4, the real indentation is 4 x 0.25 = 1 drawing unit. The default tabulator stops are 4, 8, 12, … if no tabulator stops are explicit defined.

Parameters

- **indent** (*float*) – left indentation of the first line, relative to `left`, which means an `indent` of 0 has always the same indentation as `left`
- **left** (*float*) – left indentation of the paragraph except for the first line
- **right** (*float*) – left indentation of the paragraph
- **align** – *MTextParagraphAlignment* enum
- **tab_stops** – tuple of tabulator stops, as `float` or as `str`, `float` values are left aligned tab stops, strings with prefix `"c"` are center aligned tab stops and strings with prefix `"r"` are right aligned tab stops

tostring() → str

Returns the MTEXT paragraph properties as MTEXT inline code e.g. `"\pxi-2,l2;"`.

class ezdxf.lldxf.const.**MTextParagraphAlignment**

DEFAULT

LEFT

RIGHT

CENTER

JUSTIFIED

DISTRIBUTED

Single Line Text

class ezdxf.tools.text.**TextLine**(*text: str*, *font:* AbstractFont)

Helper class which represents a single line text entity (e.g. *Text*).

Parameters

- **text** – content string
- **font** – ezdxf font definition like *MonospaceFont* or *MatplotlibFont*

property width: float

Returns the final (stretched) text width.

property height: float

Returns the final (stretched) text height.

stretch(*alignment:* TextEntityAlignment, *p1:* Vec3, *p2:* Vec3) → None

Set stretch factors for FIT and ALIGNED alignments to fit the text between *p1* and *p2*, only the distance between these points is important. Other given *alignment* values are ignore.

font_measurements() → *FontMeasurements*

Returns the scaled font measurements.

baseline_vertices(*insert:* UVec, *halign: int = 0*, *valign: int = 0*, *angle: float = 0*, *scale: tuple[float, float] = (1, 1)*) → list[*ezdxf.math._vector.Vec3*]

Returns the left and the right baseline vertex of the text line.

Parameters

- **insert** – insertion location

- **halign** – horizontal alignment left=0, center=1, right=2
- **valign** – vertical alignment baseline=0, bottom=1, middle=2, top=3
- **angle** – text rotation in radians
- **scale** – scale in x- and y-axis as 2-tuple of float

corner_vertices(*insert: UVec*, *halign: int = 0*, *valign: int = 0*, *angle: float = 0*, *scale: tuple[float, float] = (1, 1)*, *oblique: float = 0*) → list[*ezdxf.math._vector.Vec3*]

Returns the corner vertices of the text line in the order bottom left, bottom right, top right, top left.

Parameters

- **insert** – insertion location
- **halign** – horizontal alignment left=0, center=1, right=2
- **valign** – vertical alignment baseline=0, bottom=1, middle=2, top=3
- **angle** – text rotation in radians
- **scale** – scale in x- and y-axis as 2-tuple of float
- **oblique** – shear angle (slanting) in x-direction in radians

static transform_2d(*vertices: Iterable[UVec]*, *insert: UVec = Vec3(0, 0, 0)*, *shift: tuple[float, float] = (0, 0)*, *rotation: float = 0*, *scale: tuple[float, float] = (1, 1)*, *oblique: float = 0*) → list[*ezdxf.math._vector.Vec3*]

Transform any vertices from the text line located at the base location at (0, 0) and alignment LEFT.

Parameters

- **vertices** – iterable of vertices
- **insert** – insertion location
- **shift** – (shift-x, shift-y) as 2-tuple of float
- **rotation** – text rotation in radians
- **scale** – (scale-x, scale-y) as 2-tuple of float
- **oblique** – shear angle (slanting) in x-direction in radians

Functions

ezdxf.tools.text.**caret_decode**(*text: str*) → str

DXF stores some special characters using caret notation. This function decodes this notation to normalise the representation of special characters in the string.

see: https://en.wikipedia.org/wiki/Caret_notation

ezdxf.tools.text.**estimate_mtext_content_extents**(*content: str*, *font: AbstractFont*, *column_width: float = 0.0*, *line_spacing_factor: float = 1.0*) → tuple[float, float]

Estimate the width and height of the `MText` content string. The result is very inaccurate if inline codes are used or line wrapping at the column border is involved! Column breaks `\N` will be ignored.

This function uses the optional *Matplotlib* package if available.

Parameters

- **content** – the *MText* content string
- **font** – font abstraction based on *ezdxf.tools.fonts.AbstractFont*
- **column_width** – `MText.dxf.width` or 0.0 for an unrestricted column width
- **line_spacing_factor** – `MText.dxf.line_spacing_factor`

Returns
tuple[width, height]

`ezdxf.tools.text.`**estimate_mtext_extents**(*mtext:* MText) → tuple[float, float]

Estimate the width and height of a single column *MText* entity.

This function is faster than the *mtext_size()* function, but the result is very inaccurate if inline codes are used or line wrapping at the column border is involved!

This function uses the optional *Matplotlib* package if available.

Returns
Tuple[width, height]

`ezdxf.tools.text.`**fast_plain_mtext**(*text: str*, *split=False*) → list[str] | str

Returns the plain MTEXT content as a single string or a list of strings if *split* is `True`. Replaces `\P` by `\n` and removes other controls chars and inline codes.

This function is more than 4x faster than *plain_mtext()*, but does not remove single letter inline commands with arguments without a terminating semicolon like this `"\C1red text"`.

Note: Well behaved CAD applications and libraries always create inline codes for commands with arguments with a terminating semicolon like this `"\C1;red text"`!

Parameters

- **text** – MTEXT content string
- **split** – split content at line endings `\P`

`ezdxf.tools.text.`**is_text_vertical_stacked**(*text:* DXFEntity) → bool

Returns `True` if the associated text *Textstyle* is vertical stacked.

`ezdxf.tools.text.`**is_upside_down_text_angle**(*angle: float*, *tol: float = 3.0*) → bool

Returns `True` if the given text *angle* in degrees causes an upside down text in the *WCS*. The strict flip range is 90° < *angle* < 270°, the tolerance angle *tol* extends this range to: 90+tol < *angle* < 270-tol. The angle is normalized to [0, 360).

Parameters

- **angle** – text angle in degrees
- **tol** – tolerance range in which text flipping will be avoided

`ezdxf.tools.text.`**leading**(*cap_height: float*, *line_spacing: float = 1.0*) → float

Returns the distance from baseline to baseline.

Parameters

- **cap_height** – cap height of the line
- **line_spacing** – line spacing factor as percentage of 3-on-5 spacing

ezdxf.tools.text.**plain_mtext**(*text: str*, *split=False*, *tabsize: int = 4*) → list[str] | str

Returns the plain MTEXT content as a single string or a list of strings if *split* is `True`. Replaces `\P` by `\n` and removes other controls chars and inline codes.

This function is much slower than *fast_plain_mtext()*, but removes all inline codes.

Parameters

- **text** – MTEXT content string
- **split** – split content at line endings `\P`
- **tabsize** – count of replacement spaces for tabulators `^I`

ezdxf.tools.text.**plain_text**(*text: str*) → str

Returns the plain text for *Text*, *Attrib* and `Attdef` content.

ezdxf.tools.text.**safe_string**(*s: str | None*, *max_len: int = MAX_STR_LEN*) → str

Returns a string with line breaks `\n` replaced by `\P` and the length limited to *max_len*.

ezdxf.tools.text.**text_wrap**(*text: str*, *box_width: float | None*, *get_text_width: Callable[[str], float]*) → list[str]

Wrap text at `\n` and given *box_width*. This tool was developed for usage with the MTEXT entity. This isn't the most straightforward word wrapping algorithm, but it aims to match the behavior of AutoCAD.

Parameters

- **text** – text to wrap, included `\n` are handled as manual line breaks
- **box_width** – wrapping length, `None` to just wrap at `\n`
- **get_text_width** – callable which returns the width of the given string

ezdxf.tools.text.**upright_text_angle**(*angle: float*, *tol: float = 3.0*) → float

Returns a readable (upright) text angle in the range *angle* <= 90+tol or *angle* >= 270-tol. The angle is normalized to [0, 360).

Parameters

- **angle** – text angle in degrees
- **tol** – tolerance range in which text flipping will be avoided

Text Size Tools

class ezdxf.tools.text_size.**TextSize**

A frozen dataclass as return type for the *text_size()* function.

width

The text width in drawing units (float).

cap_height

The font cap-height in drawing units (float).

total_height

The font total-height = cap-height + descender-height in drawing units (float).

ezdxf.tools.text_size.**text_size**(*text: Text*) → *TextSize*

Returns the measured text width, the font cap-height and the font total-height for a *Text* entity. This function uses the optional *Matplotlib* package if available to measure the final rendering width and font-height for the Text entity as close as possible. This function does not measure the real char height! Without access to the *Matplotlib* package the *MonospaceFont* is used and the measurements are very inaccurate.

See the *text2path* add-on for more tools to work with the text path objects created by the *Matplotlib* package.

class ezdxf.tools.text_size.**MTextSize**

A frozen dataclass as return type for the *mtext_size()* function.

total_width

The total width in drawing units (float)

total_height

The total height in drawing units (float), same as max(column_heights).

column_width

The width of a single column in drawing units (float)

gutter_width

The space between columns in drawing units (float)

column_heights

A tuple of columns heights (float) in drawing units. Contains at least one column height and the column height is 0 for an empty column.

column_count

The count of columns (int).

ezdxf.tools.text_size.**mtext_size**(*mtext: MText*, *tool: MTextSizeDetector | None = None*) → *MTextSize*

Returns the total-width, -height and columns information for a *MText* entity.

This function uses the optional *Matplotlib* package if available to do font measurements and the internal text layout engine to determine the final rendering size for the MText entity as close as possible. Without access to the *Matplotlib* package the *MonospaceFont* is used and the measurements are very inaccurate.

Attention: The required full layout calculation is slow!

The first call to this function with *Matplotlib* support is very slow, because *Matplotlib* lookup all available fonts on the system. To speedup the calculation and accepting inaccurate results you can disable the *Matplotlib* support manually:

```
ezdxf.option.use_matplotlib = False
```

ezdxf.tools.text_size.**estimate_mtext_extents**(*mtext:* MText) → tuple[float, float]

Estimate the width and height of a single column *MText* entity.

This function is faster than the *mtext_size()* function, but the result is very inaccurate if inline codes are used or line wrapping at the column border is involved!

This function uses the optional *Matplotlib* package if available.

Returns
Tuple[width, height]

Fonts

The module *ezdxf.tools.fonts* manages the internal usage of fonts and has no relation how the DXF formats manages text styles.

See also:

The *Textstyle* entity, the DXF way to define fonts.

The tools in this module provide abstractions to get font measurements with and without the optional *Matplotlib* package.

For a proper text rendering the font measurements are required. *Ezdxf* has a lean approach to package dependencies, therefore the rendering results without support from the optional Matplotlib package are not very good.

Hint: If *Matplotlib* does not find an installed font and rebuilding the matplotlib font cache does not help, deleting the cache file `~/.matplotlib/fontlist-v330.json` may help.

Font Classes

ezdxf.tools.fonts.**make_font**(*ttf_path: str*, *cap_height: float*, *width_factor: float = 1.0*) → *AbstractFont*

Factory function to create a font abstraction.

Creates a *MatplotlibFont* if the Matplotlib font support is available and enabled or else a *MonospaceFont*.

Parameters

- **ttf_path** – raw font file name as stored in the *Textstyle* entity
- **cap_height** – desired cap height in drawing units.
- **width_factor** – horizontal text stretch factor

class ezdxf.tools.fonts.**AbstractFont**(*measurements:* FontMeasurements)

The *ezdxf* font abstraction.

measurement

The *FontMeasurements* data.

abstract text_width(*text: str*) → float

abstract space_width() → float

class ezdxf.tools.fonts.**MonospaceFont**(*cap_height: float*, *width_factor: float = 1.0*, *baseline: float = 0*, *descender_factor: float = DESCENDER_FACTOR*, *x_height_factor: float = X_HEIGHT_FACTOR*)

Defines a monospaced font without knowing the real font properties. Each letter has the same cap- and descender height and the same width. This font abstraction is used if no Matplotlib font support is available.

Use the *make_font()* factory function to create a font abstraction.

text_width(*text: str*) → float

Returns the text width in drawing units for the given *text* based on a simple monospaced font calculation.

space_width() → float

Returns the width of a "space" char.

class ezdxf.tools.fonts.**MatplotlibFont**(*ttf_path: str*, *cap_height: float = 1.0*, *width_factor: float = 1.0*)

This class provides proper font measurement support by using the optional Matplotlib font support.

Use the *make_font()* factory function to create a font abstraction.

text_width(*text: str*) → float

Returns the text width in drawing units for the given *text* string. Text rendering and width calculation is done by the Matplotlib `TextPath` class.

space_width() → float

Returns the width of a "space" char.

Font Anatomy

- A Visual Guide to the Anatomy of Typography: https://visme.co/blog/type-anatomy/
- Anatomy of a Character: https://www.fonts.com/content/learning/fontology/level-1/type-anatomy/anatomy

Font Properties

The default way of DXF to store fonts in the *Textstyle* entity by using the raw TTF file name is not the way how most render backends select fonts.

The render backends and web technologies select the fonts by their properties. This list shows the Matplotlib properties:

family
List of font names in decreasing order of priority. The items may include a generic font family name, either "serif", "sans-serif", "cursive", "fantasy", or "monospace".

style
"normal" ("regular"), "italic" or "oblique"

stretch
A numeric value in the range 0-1000 or one of "ultra-condensed", "extra-condensed", "condensed", "semi-condensed", "normal", "semi-expanded", "expanded", "extra-expanded" or "ultra-expanded"

weight
A numeric value in the range 0-1000 or one of "ultralight", "light", "normal", "regular", "book", "medium", "roman", "semibold", "demibold", "demi", "bold", "heavy", "extra bold", "black".

This way the backend can choose a similar font if the original font is not available.

See also:

- Matplotlib: https://matplotlib.org/stable/api/font_manager_api.html
- PyQt: https://doc.qt.io/archives/qtforpython-5.12/PySide2/QtGui/QFont.html
- W3C: https://www.w3.org/TR/2018/REC-css-fonts-3-20180920/

class ezdxf.tools.fonts.**FontFace**(*ttf*, *family*, *style*, *stretch*, *weight*)

This is the equivalent to the Matplotlib `FontProperties` class.

ttf

Raw font file name as string, e.g. "arial.ttf"

family
Family name as string, the default value is "sans-serif"

style
Font style as string, the default value is "normal"

stretch
Font stretch as string, the default value is "normal"

weight
Font weight as string, the default value is "normal"

property is_italic: bool
Returns `True` if font face is italic

property is_oblique: bool
Returns `True` if font face is oblique

property is_bold: bool
Returns `True` if font face weight > 400

class ezdxf.tools.fonts.**FontMeasurements**
See *Font Anatomy* for more information.

baseline

cap_height

x_height

descender_height

scale(*factor: float = 1.0*) → *FontMeasurements*

scale_from_baseline(*desired_cap_height: float*) → *FontMeasurements*

shift(*distance: float = 0.0*) → *FontMeasurements*

property cap_top: float

property x_top: float

property bottom: float

property total_height: float

Font Caching

Ezdxf uses Matplotlib to manage fonts and caches the collected information. The default installation of *ezdxf* provides a basic set of font properties. It is possible to create your own font cache specific for your system: see *ezdxf.options.font_cache_directory*

The font cache is loaded automatically at startup, if not disabled by setting config variable `auto_load_fonts` in `[core]` section to `False` : see *Environment Variables*

ezdxf.tools.fonts.**get_font_face**(*ttf_path: str*, *map_shx=True*) → *FontFace*

Get cached font face definition by TTF file name e.g. "Arial.ttf".

This function translates a DXF font definition by the raw TTF font file name into a *FontFace* object. Fonts which are not available on the current system gets a default font face.

Parameters

- **ttf_path** – raw font file name as stored in the *Textstyle* entity
- **map_shx** – maps SHX font names to TTF replacement fonts, e.g. "TXT" -> "txt_____.ttf"

ezdxf.tools.fonts.**get_entity_font_face**(*entity: DXFEntity*, *doc: Drawing | None = None*) → *FontFace*

Returns the *FontFace* defined by the associated text style. Returns the default font face if the *entity* does not have or support the DXF attribute "style". Supports the extended font information stored in *Textstyle* table entries.

Pass a DXF document as argument *doc* to resolve text styles for virtual entities which are not assigned to a DXF document. The argument *doc* always overrides the DXF document to which the *entity* is assigned to.

ezdxf.tools.fonts.**get_font_measurements**(*ttf_path: str*, *map_shx=True*) → *FontMeasurements*

Get cached font measurements by TTF file name e.g. "Arial.ttf".

Parameters

- **ttf_path** – raw font file name as stored in the *Textstyle* entity
- **map_shx** – maps SHX font names to TTF replacement fonts, e.g. "TXT" -> "txt_____.ttf"

ezdxf.tools.fonts.**build_system_font_cache**(*, *path=None*, *rebuild=True*) → None

Build system font cache and save it to directory *path* if given. Set *rebuild* to `False` to just add new fonts. Requires the Matplotlib package!

A rebuild has to be done only after a new ezdxf installation, or new fonts were added to your system (which you want to use), or an update of ezdxf if you don't use your own external font cache directory.

See also: *ezdxf.options.font_cache_directory*

ezdxf.tools.fonts.**load**(*path=None*, *reload=False*)

Load all caches from given *path* or from default location, defined by *ezdxf.options.font_cache_directory* or the default cache from the `ezdxf.tools` folder.

This function is called automatically at startup if not disabled by environment variable `EZDXF_AUTO_LOAD_FONTS`.

ezdxf.tools.fonts.**save**(*path=None*)

Save all caches to given *path* or to default location, defined by options.font_cache_directory or into the ezdxf.tools folder.

ACIS Tools

The *ezdxf.acis* sub-package provides some *ACIS* data management tools. The main goals of this tools are:

1. load and parse simple and known *ACIS* data structures
2. create and export simple and known *ACIS* data structures

It is NOT a goal to load and edit arbitrary existing *ACIS* structures.

Don't even try it!

These tools cannot replace the official *ACIS* SDK due to the complexity of the data structures and the absence of an *ACIS* kernel. Without access to the full documentation it is very cumbersome to reverse-engineer entities and their properties, therefore the analysis of the *ACIS* data structures is limited to the use as embedded data in DXF and DWG files.

The *ezdxf* library does not provide an *ACIS* kernel and there are no plans for implementing one because this is far beyond my capabilities, but it is possible to extract geometries made up only by flat polygonal faces (polyhedron) from ACIS data. Exporting polyhedrons as ACIS data and loading this DXF file by Autodesk products or BricsCAD works for *SAT* data for DXF R2000-R2010 and for *SAB* data for DXF R2013-R2018.

Important: Always import from the public interface module `ezdxf.acis.api`, the internal package and module structure may change in the future and imports from other modules than `api` will break.

Functions

`ezdxf.acis.api.`**`load_dxf`**(*entity: DXFEntity*) → list[*ezdxf.acis.entities.Body*]

Load the *ACIS* bodies from the given DXF entity. This is the recommended way to load ACIS data.

The DXF entity has to be an ACIS based entity and inherit from `ezdxf.entities.Body`. The entity has to be bound to a valid DXF document and the DXF version of the document has to be DXF R2000 or newer.

Raises

- ***DXFTypeError*** – invalid DXF entity type
- ***DXFValueError*** – invalid DXF document
- ***DXFVersionError*** – invalid DXF version

Warning: Only a limited count of *ACIS* entities is supported, all unsupported entities are loaded as `NONE_ENTITY` and their data is lost. Exporting such `NONE_ENTITIES` will raise an `ExportError` exception.

To emphasize that again: **It is not possible to load and re-export arbitrary ACIS data!**

Example:

```
import ezdxf
from ezdxf.acis import api as acis

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()

for e in msp.query("3DSOLID"):
    bodies = acis.load_dxf(e)
    ...
```

`ezdxf.acis.api.`**`export_dxf`**(*entity: DXFEntity*, *bodies: Sequence[*Body*]*)

Store the *ACIS* bodies in the given DXF entity. This is the recommended way to set ACIS data of DXF entities.

The DXF entity has to be an ACIS based entity and inherit from `ezdxf.entities.Body`. The entity has to be bound to a valid DXF document and the DXF version of the document has to be DXF R2000 or newer.

Raises

- ***DXFTypeError*** – invalid DXF entity type

- ***DXFValueError*** – invalid DXF document
- ***DXFVersionError*** – invalid DXF version

Example:

```
import ezdxf
from ezdxf.render import forms
from ezdxf.acis import api as acis

doc = ezdxf.new("R2000")
msp = doc.modelspace()

# create an ACIS body from a simple cube-mesh
body = acis.body_from_mesh(forms.cube())
solid3d = msp.add_3dsolid()
acis.export_dxf(solid3d, [body])
doc.saveas("cube.dxf")
```

ezdxf.acis.api.**load**(*data: str | Sequence[str] | bytes | bytearray*) → list[*ezdxf.acis.entities.Body*]

Returns a list of `Body` entities from *SAT* or *SAB* data. Accepts *SAT* data as a single string or a sequence of strings and *SAB* data as bytes or bytearray.

ezdxf.acis.api.**export_sat**(*bodies: Sequence[*Body*]*, *version: int = const.DEFAULT_SAT_VERSION*) → list[str]

Export one or more `Body` entities as text based *SAT* data.

ACIS version 700 is sufficient for DXF versions R2000, R2004, R2007 and R2010, later DXF versions require *SAB* data.

Raises

- ***ExportError*** – ACIS structures contain unsupported entities
- ***InvalidLinkStructure*** – corrupt link structure

ezdxf.acis.api.**export_sab**(*bodies: Sequence[*Body*]*, *version: int = const.DEFAULT_SAB_VERSION*) → bytes

Export one or more `Body` entities as binary encoded *SAB* data.

ACIS version 21800 is sufficient for DXF versions R2013 and R2018, earlier DXF versions require *SAT* data.

Raises

- ***ExportError*** – ACIS structures contain unsupported entities
- ***InvalidLinkStructure*** – corrupt link structure

ezdxf.acis.api.**mesh_from_body**(*body:* Body, *merge_lumps=True*) → list[ezdxf.render.mesh.MeshTransformer]

Returns a list of *MeshTransformer* instances from the given ACIS `Body` entity. The list contains multiple meshes if *merge_lumps* is `False` or just a single mesh if *merge_lumps* is `True`.

The ACIS format stores the faces in counter-clockwise orientation where the face-normal points outwards (away) from the solid body (material).

Note: This function returns meshes build up only from flat polygonal `Face` entities, for a tessellation of more complex ACIS entities (spline surfaces, tori, cones, …) is an ACIS kernel required which *ezdxf* does not provide.

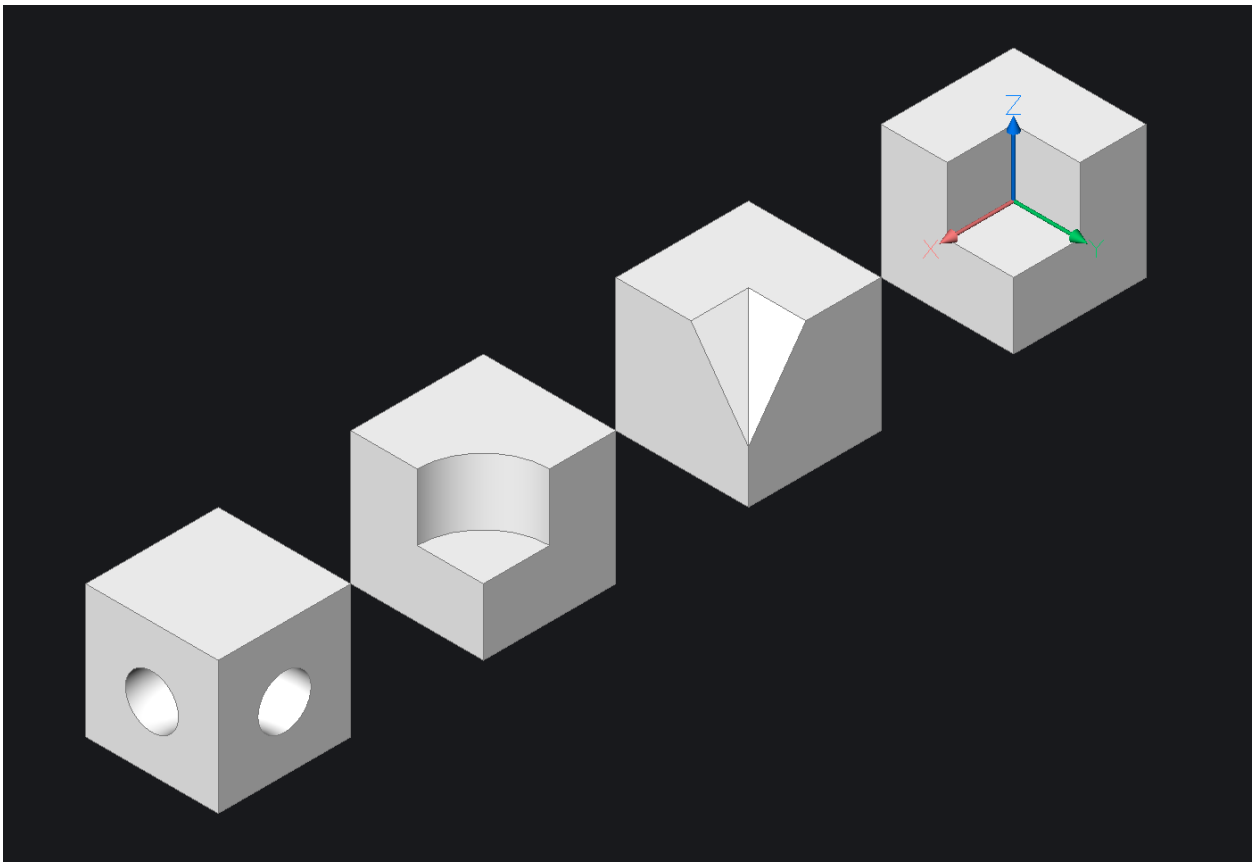
Parameters

- **body** – ACIS entity of type `Body`
- **merge_lumps** – returns all `Lump` entities from a body as a single mesh if `True` otherwise each `Lump` entity is a separated mesh

Raises

TypeError – given *body* entity has invalid type

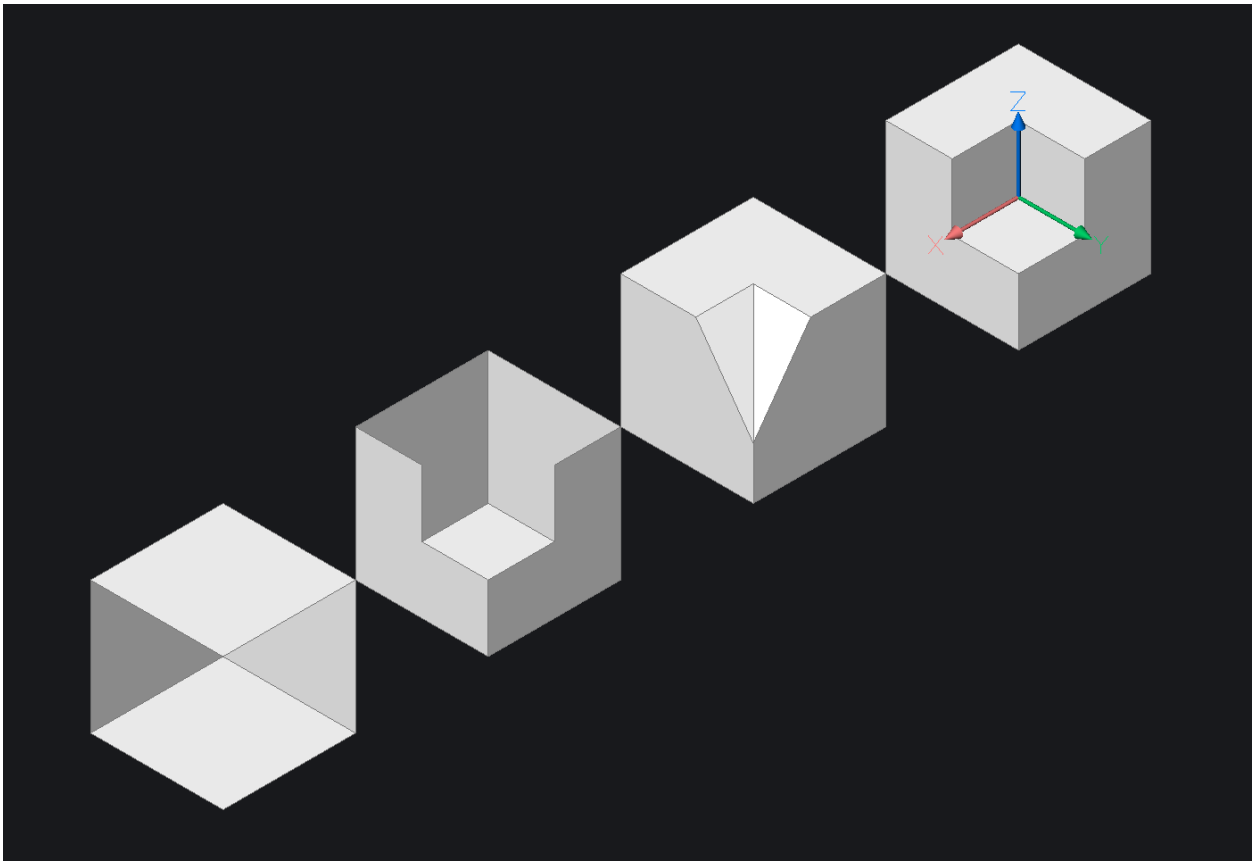
The following images show the limitations of the *mesh_from_body()* function. The first image shows the source `3DSOLID` entities with subtraction of entities with flat and curved faces:



Example script to extracts all flat polygonal faces as meshes:

```
import ezdxf
from ezdxf.acis import api as acis


doc = ezdxf.readfile("3dsolids.dxf")
msp = doc.modelspace()

doc_out = ezdxf.new()
msp_out = doc_out.modelspace()

for e in msp.query("3DSOLID"):
    for body in acis.load_dxf(data):
        for mesh in acis.mesh_from_body(body):
            mesh.render_mesh(msp_out)
doc_out.saveas("meshes.dxf")
```

The second image shows the flat faces extracted from the 3DSOLID entities and exported as *Mesh* entities:



As you can see all faces which do not have straight lines as boundaries are lost.

ezdxf.acis.api.**body_from_mesh**(*mesh: MeshBuilder*, *precision: int = 6*) → *Body*

Returns a *ACIS Body* entity from a *MeshBuilder* instance.

This entity can be assigned to a *Solid3d* DXF entity as *SAT* or *SAB* data according to the version your DXF document uses (SAT for DXF R2000 to R2010 and SAB for DXF R2013 and later).

If the *mesh* contains multiple separated meshes, each mesh will be a separated *Lump* node. If each mesh should get its own *Body* entity, separate the meshes beforehand by the method *separate_meshes*.

A closed mesh creates a solid body and an open mesh creates an open (hollow) shell. The detection if the mesh is open or closed is based on the edges of the mesh: if **all** edges of mesh have two adjacent faces the mesh is closed.

The current implementation applies automatically a vertex optimization, which merges coincident vertices into a single vertex.

Exceptions

class ezdxf.acis.api.**AcisException**

Base exception of the *ezdxf.acis* package.

class ezdxf.acis.api.**ParsingError**

Exception raised when loading invalid or unknown *ACIS* structures.

class ezdxf.acis.api.**ExportError**

Exception raised when exporting invalid or unknown *ACIS* structures.

class ezdxf.acis.api.**InvalidLinkStructure**

Exception raised when the internal link structure is damaged.

Entities

A document (sat.pdf) about the basic ACIS 7.0 file format is floating in the internet.

This section contains the additional information about the entities, I got from analyzing the SAT data extracted from DXF files exported by BricsCAD.

This documentation ignores the differences to the ACIS format prior to version 7.0 and all this differences are handled internally.

Writing support for ACIS version < 7.0 is not required because all CAD applications should be able to process version 7.0, even if embedded in a very old DXF R2000 format (tested with Autodesk TrueView, BricsCAD and Nemetschek Allplan).

The first goal is to document the entities which are required to represent a geometry as flat polygonal faces (polyhedron), which can be converted into a *MeshBuilder* object.

Topology Entities:

- *Body*
- *Lump*
- *Shell*
- *Face*
- *Loop*
- *Coedge*
- *Edge*
- *Vertex*

Geometry Entities:

- *Transform*
- *Surface*
- *Plane*
- *Curve*
- *StraightCurve*
- *Point*

ezdxf.acis.entities.**NONE_REF**

Special sentinel entity which supports the `type` attribute and the `is_none` property. Represents all unset entities. Use this idiom on any entity type to check if an entity is unset:

```
if entity.is_none:
    ...
```

AcisEntity

class ezdxf.acis.entities.**AcisEntity**

Base class for all ACIS entities.

type

Name of the type as str.

id

Unique id as int or -1 if not set.

attributes

Reference to the first `Attribute` entity (not supported).

is_none

`True` for unset entities represented by the *NONE_REF* instance.

Transform

class ezdxf.acis.entities.**Transform**(*AcisEntity*)

Represents an affine transformation operation which transform the `body` to the final location, size and rotation.

matrix

Transformation matrix of type *ezdxf.math.Matrix44*.

Body

class ezdxf.acis.entities.**Body**(*AcisEntity*)

Represents a solid geometry, which can consist of multiple *Lump* entities.

pattern

Reference to the *Pattern* entity.

lump

Reference to the first *Lump* entity

wire

Reference to the first *Wire* entity

transform

Reference to the *Transform* entity (optional)

lumps() → list[*ezdxf.acis.entities.Lump*]

Returns all linked *Lump* entities as a list.

append_lump(*lump:* Lump) → None

Append a *Lump* entity as last lump.

Pattern

class ezdxf.acis.entities.**Pattern**(*AcisEntity*)

Not implemented.

Lump

class ezdxf.acis.entities.**Lump**(*AcisEntity*)

The lump represents a connected entity and there can be multiple lumps in a *Body*. Multiple lumps are linked together by the *next_lump* attribute which points to the next lump entity the last lump has a NONE_REF as next lump. The *body* attribute references to the parent *Body* entity.

next_lump

Reference to the next *Lump* entity, the last lump references *NONE_REF*.

shell

Reference to the *Shell* entity.

body

Reference to the parent *Body* entity.

shells() → list[*ezdxf.acis.entities.Shell*]

Returns all linked *Shell* entities as a list.

append_shell(*shell:* Shell) → None

Append a *Shell* entity as last shell.

Wire

class ezdxf.acis.entities.**Wire**(*AcisEntity*)

Not implemented.

Shell

class ezdxf.acis.entities.**Shell**(*AcisEntity*)

A shell defines the boundary of a solid object or a void (subtraction object). A shell references a list of *Face* and *Wire* entities. All linked *Shell* entities are disconnected.

next_shell

Reference to the next *Shell* entity, the last shell references *NONE_REF*.

subshell

Reference to the first *Subshell* entity.

face

Reference to the first *Face* entity.

wire

Reference to the first *Wire* entity.

lump

Reference to the parent *Lump* entity.

faces() → list[*ezdxf.acis.entities.Face*]

Returns all linked *Face* entities as a list.

append_face(*face:* Face) → None

Append a *Face* entity as last face.

Subshell

class ezdxf.acis.entities.**Subshell**(*AcisEntity*)

Not implemented.

Face

class ezdxf.acis.entities.**Face**(*AcisEntity*)

A face is the building block for *Shell* entities. The boundary of a face is represented by one or more *Loop* entities. The spatial geometry of the face is defined by the *surface* object, which is a bounded or unbounded parametric 3d surface (plane, ellipsoid, spline-surface, ...).

next_face

Reference to the next *Face* entity, the last face references *NONE_REF*.

loop

Reference to the first *Loop* entity.

shell

Reference to the parent *Shell* entity.

subshell

Reference to the parent *Subshell* entity.

surface

Reference to the parametric *Surface* geometry.

sense

Boolean value of direction of the face normal with respect to the *Surface* entity:

- `True`: "reversed" direction of the face normal
- `False`: "forward" for same direction of the face normal

double_sided

Boolean value which indicates the sides of the face:

- `True`: the face is part of a hollow object and has two sides.
- `False`: the face is part of a solid object and has only one side which points away from the "material".

containment

Unknown meaning.

If *double_sided* is `True`:

- `True` is "in"
- `False` is "out"

loops() → list[*ezdxf.acis.entities.Loop*]

Returns all linked *Loop* entities as a list.

append_loop(*loop:* Loop) → None

Append a *Loop* entity as last loop.

Loop

class ezdxf.acis.entities.**Loop**(*AcisEntity*)

A loop represents connected coedges which are building the boundaries of a *Face*, there can be multiple loops for a single face e.g. faces can contain holes. The *coedge* attribute references the first *Coedge* of the loop, the additional coedges are linked to this first *Coedge*. In closed loops the coedges are organized as a circular list, in open loops the last coedge references the *NONE_REF* entity as `next_coedge` and the first coedge references the *NONE_REF* as `prev_coedge`.

next_loop

Reference to the next *Loop* entity, the last loop references *NONE_REF*.

coedge

Reference to the first *Coedge* entity.

face

Reference to the parent *Face* entity.

coedges() → list[*ezdxf.acis.entities.Coedge*]

Returns all linked *Coedge* entities as a list.

set_coedges(*coedges: list[*ezdxf.acis.entities.Coedge*]*, *close=True*) → None

Set all coedges of a loop at once.

Coedge

class ezdxf.acis.entities.**Coedge**(*AcisEntity*)

The coedges are a double linked list where *next_coedge* points to the next *Coedge* and *prev_coedge* to the previous *Coedge*.

The *partner_coedge* attribute references the first partner *Coedge* of an adjacent *Face*, the partner edges are organized as a circular list. In a manifold closed surface each *Face* is connected to one partner face by an *Coedge*. In a non-manifold surface a face can have more than one partner face.

next_coedge

References the next *Coedge*, reference the *NONE_REF* if it is the last coedge in an open *Loop*.

prev_coedge

References the previous *Coedge*, reference the *NONE_REF* if it is the first coedge in an open *Loop*.

partner_coedge

References the partner *Coedge* of an adjacent *Face* entity. The partner coedges are organized in a circular list.

edge

References the *Edge* entity.

loop

References the parent *Loop* entity.

pcurve

References the *PCurve* entity.

Edge

class ezdxf.acis.entities.**Edge**(*AcisEntity*)

The *Edge* entity represents the physical edge of an object. Its geometry is defined by the bounded portion of a parametric space curve. This bounds are stored as object-space *Vertex* entities.

start_vertex

The start *Vertex* of the space-curve in object coordinates, if *NONE_REF* the curve is unbounded in this direction.

start_param

The parametric starting bound for the parametric curve. Evaluating the *curve* for this parameter should return the coordinates of the *start_vertex*.

end_vertex

The end *Vertex* of the space-curve in object coordinates, if *NONE_REF* the curve is unbounded in this direction.

end_param

The parametric end bound for the parametric curve.

coedge

Parent *Coedge* of this edge.

curve

The parametric space-curve which defines this edge. The curve can be the NULL_REF while both *Vertex* entities are the same vertex. In this case the *Edge* represents an single point like the apex of a cone.

sense

Boolean value which indicates the direction of the edge:

- True: the edge has the "reversed" direction as the underlying curve
- False: the edge has the same direction as the underlying curve ("forward")

convexity

Unknown meaning, always the string "unknown".

Vertex

class ezdxf.acis.entities.**Vertex**(*AcisEntity*)

Represents a vertex of an *Edge* entity and references a *Point* entity.

point

The spatial location in object-space as *Point* entity.

edge

Parent *Edge* of this vertex. The vertex can be referenced by multiple edges, anyone of them can be the parent of the vertex.

Surface

class ezdxf.acis.entities.**Surface**(*AcisEntity*)

Abstract base class for all parametric surfaces.

The parameterization of any *Surface* maps a 2D rectangle (u, v parameters) into the spatial object-space (x, y, z).

u_bounds

Tuple of (start bound, end bound) parameters as two floats which define the bounds of the parametric surface in the u-direction, one or both values can be `math.inf` which indicates an unbounded state of the surface in that direction.

v_bounds

Tuple of (start bound, end bound) parameters as two floats which define the bounds of the parametric surface in the v-direction, one or both values can be `math.inf` which indicates an unbounded state of the surface in that direction.

abstract evaluate(*u: float*, *v: float*) → *Vec3*

Returns the spatial location at the parametric surface for the given parameters *u* and *v*.

Plane

class ezdxf.acis.entities.**Plane**(*Surface*)

Defines a flat plan as parametric surface.

origin

Location vector of the origin of the flat plane as *Vec3*.

normal

Normal vector of the plane as *Vec3*. Has to be an unit-vector!

u_dir

Direction vector of the plane in u-direction as *Vec3*. Has to be an unit-vector!

v_dir

Direction vector of the plane in v-direction as *Vec3*. Has to be an unit-vector!

reverse_v

Boolean value which indicates the orientation of the coordinate system:

- `True`: left-handed system, the v-direction is reversed and the normal vector is *v_dir* cross *u_dir*.
- `False`: right-handed system and the normal vector is *u_dir* cross *v_dir*.

Curve

class ezdxf.acis.entities.**Curve**(*AcisEntity*)

Abstract base class for all parametric curves.

The parameterization of any *Curve* maps a 1D line (the parameter) into the spatial object-space (x, y, z).

bounds

Tuple of (start bound, end bound) parameters as two floats which define the bounds of the parametric curve, one or both values can be `math.inf` which indicates an unbounded state of the curve in that direction.

abstract evaluate(*param: float*) → *Vec3*

Returns the spatial location at the parametric curve for the given parameter.

StraightCurve

class ezdxf.acis.entities.**StraightCurve**(*Curve*)

Defines a straight line as parametric curve.

origin

Location vector of the origin of the straight line as *Vec3*.

direction

Direction vector the straight line as *Vec3*. Has to be an unit-vector!

PCurve

class ezdxf.acis.entities.**PCurve**(*AcisEntity*)

Not implemented.

Point

class ezdxf.acis.entities.**Point**(*AcisEntity*)

Represents a point in the 3D object-space.

location

Cartesian coordinates as *Vec3*.

6.9.11 Global Options

Global Options Object

The global *ezdxf* options are stored in the object *ezdxf.options*.

Recommended usage of the global `options` object:

```
import ezdxf

value = ezdxf.options.attribute
```

The `options` object uses the Standard Python class `ConfigParser` to manage the configuration. Shortcut attributes like *test_files* are simple properties and most shortcuts are read only marked by (Read only), read and writeable attributes are marked by (Read/Write).

To change options, especially the read only attributes, you have to edit the config file with a text editor, or set options by the *set()* method and write the current configuration into a config file.

Config Files

The default config files are loaded from the user home directory as "~/.config/ezdxf/ezdxf.ini", and the current working directory as "./ezdxf.ini". A custom config file can be specified by the environment variable `EZDXF_CONFIG_FILE`. Ezdxf follows the XDG Base Directory specification if the environment variable `XDG_CONFIG_HOME` is set.

The config file loading order:

1. user home directory: "~/.config/ezdxf/ezdxf.ini"
2. current working directory: "./ezdxf.ini"
3. config file specified by `EZDXF_CONFIG_FILE`

A configuration file that is loaded later does not replace the previously loaded ones, only the existing options in the newly loaded file are added to the configuration and can overwrite existing options.

Configuration files are regular INI files, managed by the standard Python ConfigParser class.

File Structure:

```
[core]
default_dimension_text_style = OpenSansCondensed-Light
test_files = D:\Source\dxftest
font_cache_directory =
load_proxy_graphics = true
store_proxy_graphics = true
log_unprocessed_tags = false
filter_invalid_xdata_group_codes = true
write_fixed_meta_data_for_testing = false
disable_c_ext = false

[browse-command]
text_editor = "C:\Program Files\Notepad++\notepad++.exe" "{filename}" -n{num}
```

Modify and Save Changes

This code shows how to get and set values of the underlying `ConfigParser` object, but use the shortcut attributes if available:

```
# Set options, value has to ba a str, use "true"/"false" for boolean values
ezdxf.options.set(section, key, value)

# Get option as string
value = ezdxf.options.get(section, key, default="")

# Special getter for boolean, int and float
value = ezdxf.options.get_bool(section, key, default=False)
value = ezdxf.options.get_int(section, key, default=0)
value = ezdxf.options.get_float(section, key, default=0.0)
```

If you set options, they are not stored automatically in a config file, you have to write back the config file manually:

```
# write back the default user config file "ezdxf.ini" in the
# user home directory
ezdxf.options.write_home_config()

# write back to the default config file "ezdxf.ini" in the
```

(continues on next page)

(continued from previous page)

```
# current working directory
ezdxf.options.write_file()

# write back to a specific config file
ezdxf.options.write_file("my_config.ini")
# which has to be loaded manually at startup
ezdxf.options.read_file("my_config.ini")
```

This example shows how to change the `test_files` path and save the changes into a custom config file "my_config.ini":

```
import ezdxf

test_files = Path("~/my-dxf-test-files").expand_user()
ezdxf.options.set(
    ezdxf.options.CORE,  # section
    "test_files",  # key
    "~/my-dxf-test-files",  # value
)
ezdxf.options.write_file("my_config.ini")
```

Use a Custom Config File

You can specify a config file by the environment variable `EZDXF_CONFIG_FILE`, which is loaded after the default config files.

```
C:\> set EZDXF_CONFIG_FILE=D:\user\path\custom.ini
```

Custom config files are not loaded automatically like the default config files.

This example shows how to load the previous created custom config file "my_config.ini" from the current working directory:

```
import ezdxf

ezdxf.options.read("my_config.ini")
```

That is all and because this is the last loaded config file, it overrides all default config files and the config file specified by `EZDXF_CONFIG_FILE`.

Functions

`ezdxf.options.`**`set`**(*section: str*, *key: str*, *value: str*)

Set option *key* in *section* to *values* as `str`.

`ezdxf.options.`**`get`**(*section: str*, *key: str*, *default: str = ''*) → str

Get option *key* in *section* as string.

`ezdxf.options.`**`get_bool`**(*section: str*, *key: str*, *default: bool = False*) → bool

Get option *key* in *section* as `bool`.

`ezdxf.options.`**`get_int`**(*section: str*, *key: str*, *default: int = 0*) → int

Get option *key* in *section* as `int`.

ezdxf.options.**get_float**(*section: str*, *key: str*, *default: float = 0.0*) → flot

Get option *key* in *section* as `float`.

ezdxf.options.**write**(*fp: TextIO*)

Write configuration into given file object *fp*, the file object must be a writeable text file with "utf8" encoding.

ezdxf.options.**write_file**(*filename: str = 'ezdxf.ini'*)

Write current configuration into file *filename*, default is "ezdxf.ini" in the current working directory.

ezdxf.options.**write_home_config**()

Write configuration into file "~/.config/ezdxf/ezdxf.ini", `$XDG_CONFIG_HOME` is supported if set.

ezdxf.options.**read_file**(*filename: str*)

Append content from config file *filename*, but does not reset the configuration.

ezdxf.options.**print**()

Print configuration to *stdout*.

ezdxf.options.**reset**()

Reset options to factory default values.

ezdxf.options.**delete_default_config_files**()

Delete the default config files "ezdxf.ini" in the current working and in the user home directory "~/.config/ezdxf", `$XDG_CONFIG_HOME` is supported if set.

ezdxf.options.**preserve_proxy_graphics**(*state=True*)

Enable/disable proxy graphic load/store support by setting the options `load_proxy_graphics` and `store_proxy_graphics` to *state*.

ezdxf.options.**loaded_config_files**

Read only property of loaded config files as tuple for `Path` objects.

Core Options

For all core options the section name is `core`.

Default Dimension Text Style

The default dimension text style is used by the DIMENSION renderer of *ezdxf*, if the specified text style exist in the STYLE table. To use any of the default style of *ezdxf* you have to setup the styles at the creation of the DXF document: `ezdxf.new(setup=True)`, or setup the *ezdxf* default styles for a loaded DXF document:

```
import ezdxf
from ezdxf.tool.standard import setup_drawing

doc = ezdxf.readfile("your.dxf")
setup_drawing(doc)
```

Config file key: `default_dimension_text_style`

Shortcut attribute:

ezdxf.options.**default_dimension_text_style**

(Read/Write) Get/Set default text style for DIMENSION rendering, default value is `OpenSansCondensed-Light`.

Font Cache Directory

Ezdxf has a bundled font cache to have faster access to font metrics. This font cache includes only fonts installed on the developing workstation. To add the fonts of your computer to this cache, you have to create your own external font cache. This has to be done only once after *ezdxf* was installed, or to add new installed fonts to the cache, and this requires the *Matplotlib* package.

This example shows, how to create an external font cache in the recommended directory of the XDG Base Directory specification: `"~/.cache/ezdxf"`.

```
import ezdxf
from ezdxf.tools import fonts

# xdg_path() returns "$XDG_CACHE_HOME/ezdxf" or "~/.cache/ezdxf" if
# $XDG_CACHE_HOME is not set
font_cache_dir = ezdxf.options.xdg_path("XDG_CACHE_HOME", ".cache")
fonts.build_system_font_cache(path=font_cache_dir)
ezdxf.options.font_cache_directory = font_cache_dir
# Save changes to the default config file "~/.config/ezdxf/ezdxf.ini"
# to load the font cache always from the new location.
ezdxf.options.write_home_config()
```

Config file key: `font_cache_directory`

Shortcut attribute:

`ezdxf.options.`**`font_cache_directory`**

(Read/Write) Get/set the font cache directory, if the directory is an empty string, the bundled font cache is used. Expands "~" construct automatically.

Load Proxy Graphic

Proxy graphics are not essential for DXF files, but they can provide a simple graphical representation for complex entities, but extra memory is needed to store this information. You can save some memory by not loading the proxy graphic, but the proxy graphic is lost if you write back the DXF file.

The current version of *ezdxf* uses this proxy graphic to render MLEADER entities by the *drawing* add-on.

Config file key: `load_proxy_graphics`

Shortcut attribute:

`ezdxf.options.`**`load_proxy_graphics`**

(Read/Write) Load proxy graphics if `True`, default is `True`.

Store Proxy Graphic

Prevent exporting proxy graphics if set to `False`.

Config file key: `store_proxy_graphics`

Shortcut attribute:

`ezdxf.options.`**`store_proxy_graphics`**

(Read/Write) Export proxy graphics if `True`, default is `True`.

Support Directories

Search directories for support files:

- plot style tables, the CTB or STB pen assignment files
- shape font files of type SHX or SHP

Config file key: `support_dirs`

Shortcut attribute:

`ezdxf.options.`**`support_dirs`**

(Read/Write) Search directories as list of strings.

Debugging Options

For all debugging options the section name is `core`.

Test Files

Path to test files. Some of the CADKit test files are used by the integration tests, these files should be located in the `ezdxf.options.test_files_path / "CADKitSamples"` folder.

Config file key: `test_files`

Shortcut attributes:

`ezdxf.options.`**`test_files`**

(Read only) Returns the path to the *ezdxf* test files as `str`, expands "~" construct automatically.

`ezdxf.options.`**`test_files_path`**

(Read only) Path to test files as `pathlib.Path` object.

Filter Invalid XDATA Group Codes

Only a very limited set of group codes is valid in the XDATA section and AutoCAD is very picky about that. *Ezdxf* removes invalid XDATA group codes if this option is set to `True`, but this needs processing time, which is wasted if you get your DXF files from trusted sources like AutoCAD or BricsCAD.

Config file key: `filter_invalid_xdata_group_codes`

`ezdxf.options.`**`filter_invalid_xdata_group_codes`**

(Read only) Filter invalid XDATA group codes, default value is `True`.

Log Unprocessed Tags

Logs unprocessed DXF tags, this helps to find new and undocumented DXF features.

Config file key: `log_unprocessed_tags`

`ezdxf.options.`**`log_unprocessed_tags`**

(Read/Write) Log unprocessed DXF tags for debugging, default value is `False`.

Write Fixed Meta Data for Testing

Write the DXF files with fixed meta data to test your DXF files by a diff-like command, this is necessary to get always the same meta data like the saving time stored in the HEADER section. This may not work across different *ezdxf* versions!

Config file key: `write_fixed_meta_data_for_testing`

`ezdxf.options.`**`write_fixed_meta_data_for_testing`**

(Read/Write) Enable this option to always create same meta data for testing scenarios, e.g. to use a diff-like tool to compare DXF documents, default is `False`.

Disable C-Extension

It is possible to deactivate the optional C-extensions if there are any issues with the C-extensions. This has to be done in a default config file or by environment variable before the first import of *ezdxf*. For `pypy3` the C-extensions are always disabled, because the JIT compiled Python code is much faster.

Important: This option works only in the **default config files**, user config files which are loaded by *`ezdxf.options.read_file()`* cannot disable the C-Extensions, because at this point the setup process of *ezdxf* is already finished!

Config file key: `disable_c_ext`

`ezdxf.options.`**`disable_c_ext`**

(Read only) This option disables the C-extensions if `True`. This can only be done before the first import of *ezdxf* by using a config file or the environment variable `EZDXF_DISABLE_C_EXT`.

Use C-Extensions

`ezdxf.options.`**`use_c_ext`**

(Read only) Shows the actual state of C-extensions usage.

Use Matplotlib

This option can deactivate Matplotlib support for testing. This option is not stored in the `ConfigParser` object and is therefore not supported by config files!

Only access by attribute is supported:

`ezdxf.options.`**`use_matplotlib`**

(Read/Write) Activate/deactivate Matplotlib support (e.g. for testing) if Matplotlib is installed, otherwise *`use_matplotlib`* is always `False`.

Environment Variables

Some feature can be controlled by environment variables. Command line example for disabling the optional C-extensions on Windows:

```
C:\> set EZDXF_DISABLE_C_EXT=1
```

Important: If you change any environment variable, you have to restart the Python interpreter!

EZDXF_DISABLE_C_EXT
Set environment variable `EZDXF_DISABLE_C_EXT` to `1` or `True` to disable the usage of the C-extensions.

EZDXF_TEST_FILES
Path to the *ezdxf* test files required by some tests, for instance the CADKit sample files should be located in the `EZDXF_TEST_FILES/CADKitSamples` folder. See also option *ezdxf.options.test_files*.

EZDXF_CONFIG_FILE
Specifies a user config file which will be loaded automatically after the default config files at the first import of *ezdxf*.

6.9.12 Miscellaneous

Zoom Layouts

These functions mimic the ZOOM commands in CAD applications.

Zooming means resetting the current viewport limits to new values. The coordinates for the functions *center()* and *window()* are drawing units for the model space and paper space units for paper space layouts. The modelspace units in `Drawing.units` are ignored.

The extents detection for the functions `entities()` and *extents()* is done by the *ezdxf.bbox* module. Read the associated documentation to understand the limitations of the *ezdxf.bbox* module. Tl;dr The extents detection is **slow** and **not accurate**.

Because the ZOOM operations in CAD applications are not that precise, then zoom functions of this module uses the fast bounding box calculation mode of the `bbox` module, which means the argument *flatten* is always `False` for *extents()* function calls.

The region displayed by CAD applications also depends on the aspect ratio of the application window, which is not available to *ezdxf*, therefore the viewport size is just an educated guess of an aspect ratio of 2:1 (16:9 minus top toolbars and the bottom status bar).

Warning: All zoom functions replace the current viewport configuration by a single window configuration.

Example to reset the main CAD viewport of the model space to the extents of its entities:

```
import ezdxf
from ezdxf import zoom

doc = ezdxf.new()
msp = doc.modelspace()
... # add your DXF entities
```

(continues on next page)

(continued from previous page)

```
zoom.extents(msp)
doc.saveas("your.dxf")
```

ezdxf.zoom.**center**(*layout: Layout*, *point: Sequence[float]* | Vec2 | Vec3, *size: Sequence[float]* | Vec2 | Vec3)

Resets the active viewport center of *layout* to the given *point*, argument *size* defines the width and height of the viewport. Replaces the current viewport configuration by a single window configuration.

ezdxf.zoom.**objects**(*layout: Layout*, *entities: Iterable[DXFEntity]*, *factor: float = 1*)

Resets the active viewport limits of *layout* to the extents of the given *entities*. Only entities in the given *layout* are taken into account. The argument *factor* scales the viewport limits. Replaces the current viewport configuration by a single window configuration.

ezdxf.zoom.**extents**(*layout: Layout*, *factor: float = 1*)

Resets the active viewport limits of *layout* to the extents of all entities in this *layout*. The argument *factor* scales the viewport limits. Replaces the current viewport configuration by a single window configuration.

ezdxf.zoom.**window**(*layout: Layout*, *p1: Sequence[float]* | Vec2 | Vec3, *p2: Sequence[float]* | Vec2 | Vec3)

Resets the active viewport limits of *layout* to the lower left corner *p1* and the upper right corner *p2*. Replaces the current viewport configuration by a single window configuration.

Load DXF Comments

ezdxf.comments.**from_stream**(*stream: TextIO*, *codes: set[int]* | *None* = *None*) → Iterable[*DXFTag*]

Yields comment tags from text *stream* as *DXFTag* objects.

Parameters

- **stream** – input text stream
- **codes** – set of group codes to yield additional DXF tags e.g. {5, 0} to also yield handle and structure tags

ezdxf.comments.**from_file**(*filename: str*, *codes: set[int]* | *None* = *None*) → Iterable[*DXFTag*]

Yields comment tags from file *filename* as *DXFTag* objects.

Parameters

- **filename** – filename as string
- **codes** – yields also additional tags with specified group codes e.g. {5, 0} to also yield handle and structure tags

6.10 Launcher

The command line script *ezdxf* launches various sub-commands:

pp	DXF pretty printer, replacement for the previous *dxfpp* command
audit	Audit and repair DXF files
draw	Draw and convert DXF files by the Matplotlib backend
view	PyQt DXF file viewer
pillow	Draw and convert DXF files by the Pillow backend
browse	PyQt DXF structure browser for DXF debugging and curious people
browse-acis	PyQt ACIS entity content browser for SAT/SAB debugging
strip	Strip comments and THUMBNAILIMAGE section from DXF files
config	Manage config files
info	Show information and optional stats of DXF files as loaded by ezdxf

The help option `-h` is supported by the main script and all sub-commands:

```
C:\> ezdxf -h
usage: ezdxf [-h] [-V] [-v] [--config CONFIG] [--log LOG]
             {pp,audit,draw,view,browse,browse-acis,strip,config} ...

Command launcher for the Python package "ezdxf":
https://pypi.org/project/ezdxf/

positional arguments:
  {pp,audit,draw,view,browse,strip}
    pp                  pretty print DXF files as HTML file
    audit               audit and repair DXF files
    draw                draw and convert DXF files by Matplotlib
    view                view DXF files by the PyQt viewer
    pillow              draw and convert DXF files by Pillow
    browse              browse DXF file structure
    browse-acis         browse ACIS structures in DXF files
    strip               strip comments from DXF files
    config              manage config files
    info                show information and optional stats of DXF files loaded by␣
↪ezdxf,
                        this may not represent the original content of the file, use␣
↪the
                        browse command to see the original content

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show version and exit
  -v, --verbose         give more output
  --config CONFIG       path to a config file
  --log LOG             path to a verbose appending log
```

Note: The `ezdxf` script is the only executable script installed on the user system.

6.10.1 Pretty Printer

Pretty print the DXF text content as HTML file and open the file in the default web browser:

```
C:\> ezdxf pp -o gear.dxf
```

Print help:

```
C:\> ezdxf pp -h
usage: ezdxf pp [-h] [-o] [-r] [-x] [-l] [-s SECTIONS] FILE [FILE ...]

positional arguments:
  FILE                  DXF files pretty print

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -o, --open            open generated HTML file by the default web browser
  -r, --raw             raw mode, no DXF structure interpretation
  -x, --nocompile       don't compile points coordinates into single tags (only in␣
↪raw mode)
  -l, --legacy          legacy mode, reorder DXF point coordinates
  -s SECTIONS, --sections SECTIONS
                        choose sections to include and their order, h=HEADER,␣
↪c=CLASSES,
                        t=TABLES, b=BLOCKS, e=ENTITIES, o=OBJECTS
```

6.10.2 Audit

Audit and recover the DXF file "gear.dxf" and save the recovered version as "gear.rec.dxf":

```
C:\> ezdxf audit -s gear.dxf

auditing file: gear.dxf
No errors found.
Saved recovered file as: gear.rec.dxf
```

Print help:

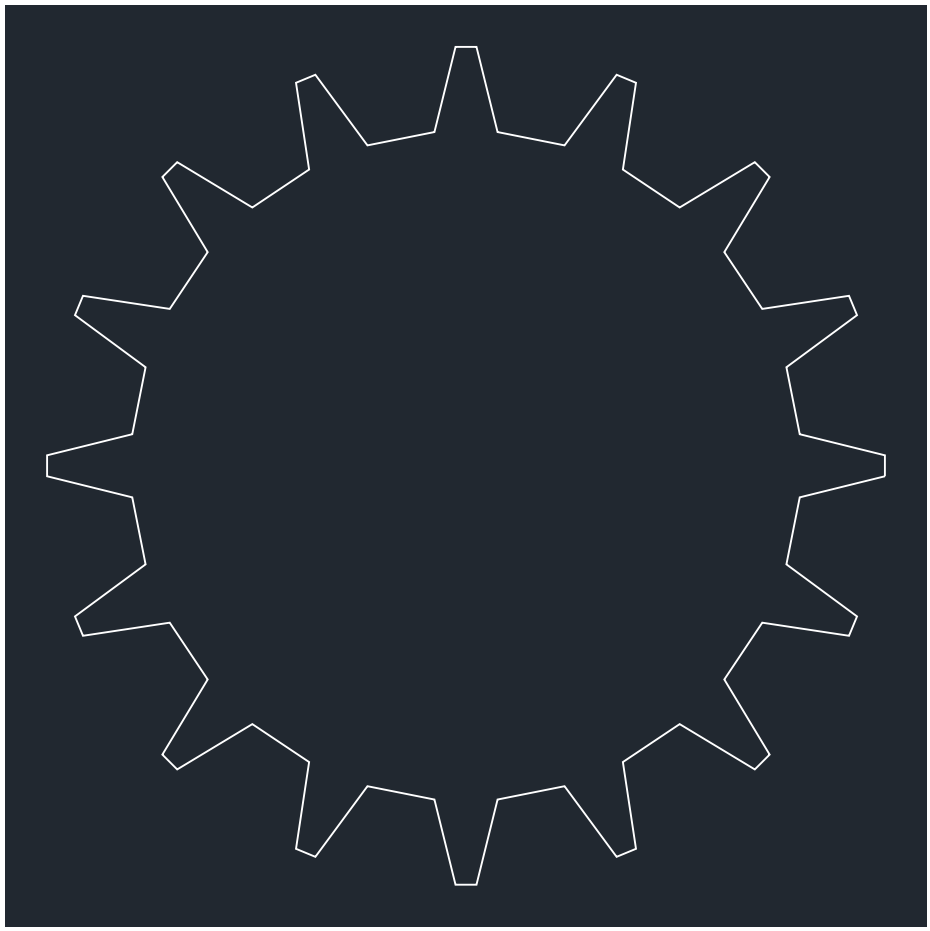
```
C:\> ezdxf audit -h
usage: ezdxf audit [-h] [-s] FILE [FILE ...]

positional arguments:
  FILE        audit DXF files

optional arguments:
  -h, --help  show this help message and exit
  -s, --save  save recovered files with extension ".rec.dxf"
```

6.10.3 Draw

Convert the DXF file "gear.dxf" into a SVG file by the *Matplotlib* backend:

```
C:\> ezdxf draw -o gear.pdf gear.dxf
```

The "gear.pdf" created by the *Matplotlib* backend:



Show all output formats supported by the *Matplotlib* backend on your system. This output may vary:

```
C:\> ezdxf draw --formats
eps: Encapsulated Postscript
jpg: Joint Photographic Experts Group
jpeg: Joint Photographic Experts Group
pdf: Portable Document Format
pgf: PGF code for LaTeX
png: Portable Network Graphics
ps: Postscript
raw: Raw RGBA bitmap
rgba: Raw RGBA bitmap
svg: Scalable Vector Graphics
svgz: Scalable Vector Graphics
tif: Tagged Image File Format
tiff: Tagged Image File Format
```

Print help:

```
C:\> ezdxf draw -h
usage: ezdxf draw [-h] [--formats] [-l LAYOUT] [--all-layers-visible]
                  [--all-entities-visible] [-o OUT] [--dpi DPI] [-v]
                  [FILE]

positional arguments:
  FILE                  DXF file to view or convert
```

(continues on next page)

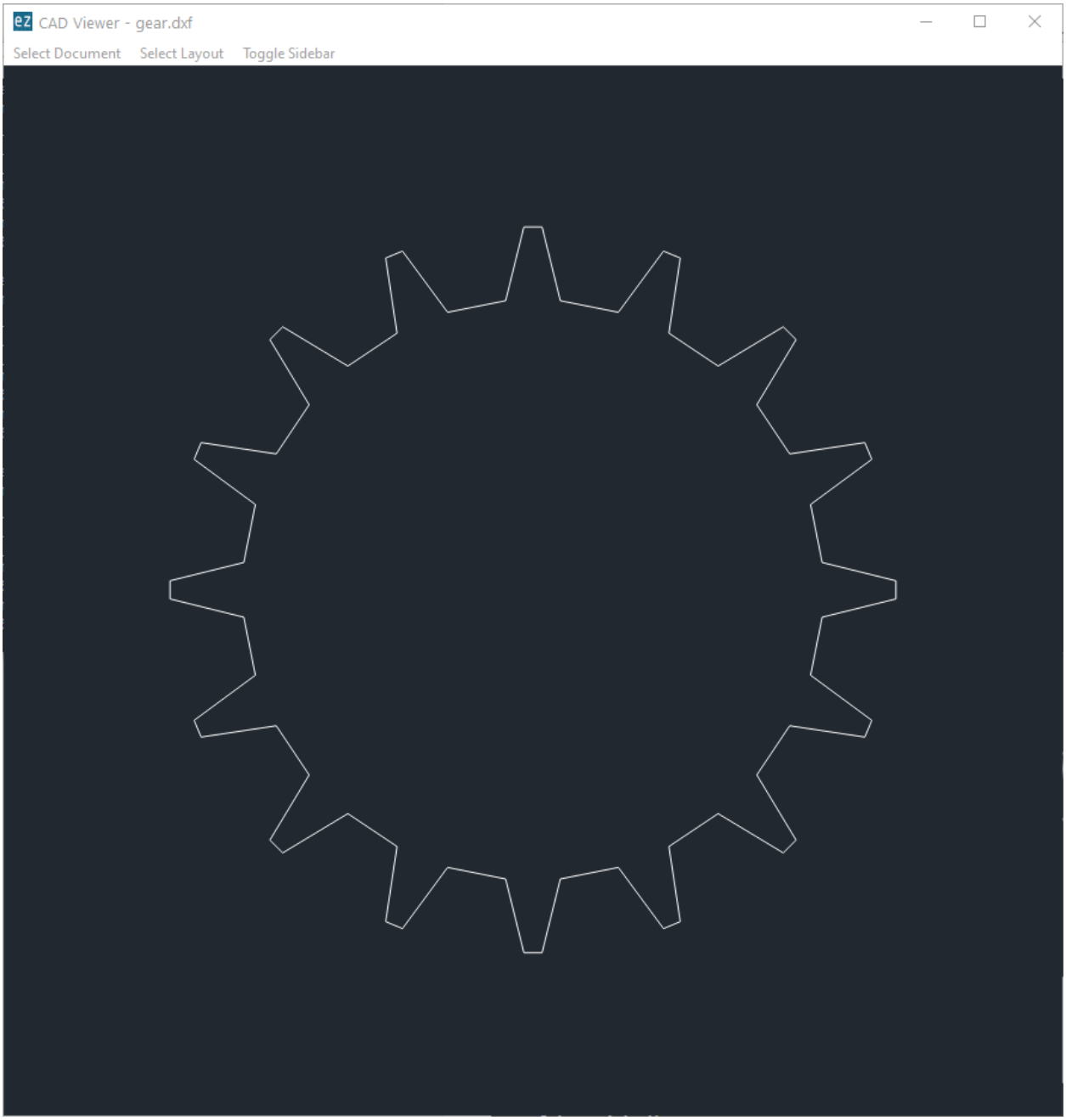
(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  --formats             show all supported export formats and exit
  -l LAYOUT, --layout LAYOUT
                        select the layout to draw, default is "Model"
  --all-layers-visible  draw all layers including the ones marked as invisible
  --all-entities-visible
                        draw all entities including the ones marked as
                        invisible (some entities are individually marked as
                        invisible even if the layer is visible)
  -o OUT, --out OUT     output filename for export
  --dpi DPI             target render resolution, default is 300
  -v, --verbose         give more output
```

6.10.4 View

View the DXF file "gear.dxf" by the *PyQt* backend:

```
C:\> ezdxf view gear.dxf
```



Print help:

```
C:\> ezdxf view -h
usage: ezdxf view [-h] [-l LAYOUT] [--lwscale LWSCALE] [FILE]

positional arguments:
  FILE                  DXF file to view

optional arguments:
  -h, --help            show this help message and exit
  -l LAYOUT, --layout LAYOUT
                        select the layout to draw, default is "Model"
```

(continues on next page)

(continued from previous page)

```
  --lwscale LWSCALE     set custom line weight scaling, default is 0 to
                        disable line weights at all
```

6.10.5 Pillow

Convert the DXF file "gear.dxf" into a PNG file by the *Pillow* backend:

```
C:\> ezdxf pillow -o gear.png gear.dxf
```

Advantage over the *Draw* command is the speed and much less memory usage, disadvantage is the lower text rendering quality. The speed advantages is lost for the text modes OUTLINE and FILLED, because the text-path rendering is done by *Matplotlib*, but the advantage of the lower memory consumption remains.

Print help:

```
C:\> ezdxf pillow -h
usage: ezdxf pillow [-h] [-o OUT] [-l LAYOUT] [-i IMAGE_SIZE] [-b BACKGROUND]
                    [-r OVERSAMPLING] [-m MARGIN] [-t {0,1,2,3}] [--dpi DPI] [-v]
                    [FILE]

positional arguments:
  FILE                  DXF file to draw

options:
  -h, --help            show this help message and exit
  -o OUT, --out OUT     output filename, the filename extension defines the image␣
↪format
                        (.png, .jpg, .tif, .bmp, ...)
  -l LAYOUT, --layout LAYOUT
                        name of the layout to draw, default is "Model"
  -i IMAGE_SIZE, --image_size IMAGE_SIZE
                        image size in pixels as "width,height", default is "1920,1080
↪",
                        supports also "x" as delimiter like "1920x1080". A single
                        integer is used for both directions e.g. "2000" defines an␣
↪image
                        size of 2000x2000. The image is centered for the smaller DXF
                        drawing extent.
  -b BACKGROUND, --background BACKGROUND
                        override background color in hex format "RRGGBB" or "RRGGBBAA
↪",
                        e.g. use "FFFFFF00" to get a white transparent background and␣
↪a
                        black foreground color (ACI=7), because a light background␣
↪gets
                        a black foreground color or vice versa "00000000" for a black
                        transparent background and a white foreground color.
  -r OVERSAMPLING, --oversampling OVERSAMPLING
                        oversampling factor, default is 2, use 0 or 1 to disable
                        oversampling
  -m MARGIN, --margin MARGIN
                        minimal margin around the image in pixels, default is 10
  -t {0,1,2,3}, --text-mode {0,1,2,3}
                        text mode: 0=ignore, 1=placeholder, 2=outline, 3=filled,␣
↪default
```

(continues on next page)

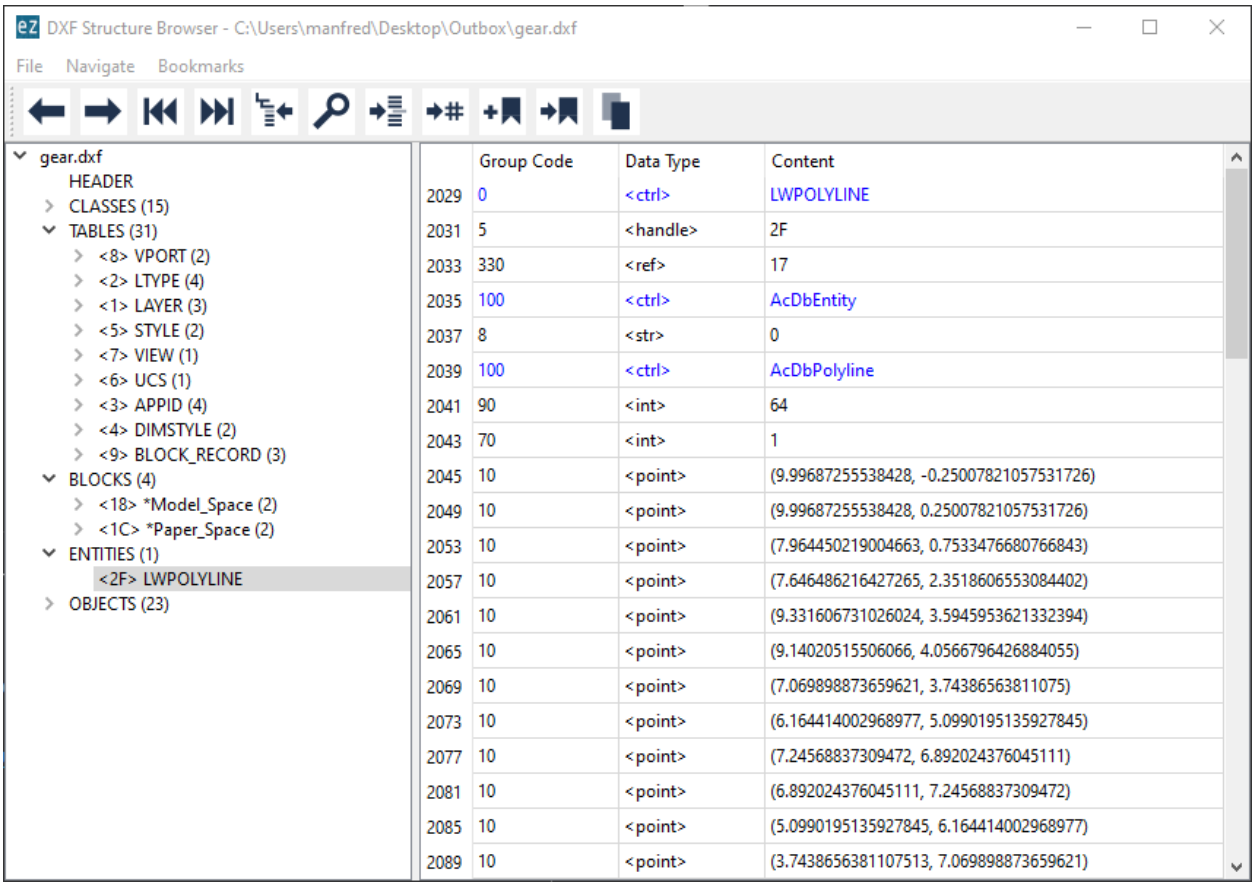
(continued from previous page)

```
                        is 2
  --dpi DPI             output resolution in pixels/inch which is significant for the
                        linewidth, default is 300
  -v, --verbose         give more output
```

6.10.6 Browse

Browse the internal structure of a DXF file like a file system:

```
C:\> ezdxf browse gear.dxf
```




```
C:\> ezdxf browse -h
usage: ezdxf browse [-h] [-l LINE] [-g HANDLE] [FILE]

positional arguments:
  FILE                  DXF file to browse

optional arguments:
  -h, --help            show this help message and exit
  -l LINE, --line LINE  go to line number
  -g HANDLE, --handle HANDLE
                        go to entity by HANDLE, HANDLE has to be a hex value without
                        any prefix like 'fefe'
```

The *browse* command stores options in the config file, e.g. for the *Notepad++* on Windows:

```
[browse-command]

text_editor = "C:\Program Files\Notepad++\notepad++.exe" "{filename}" -n{num}
icon_size = 32
```

`text_editor` is a simple format string: `text_editor.format(filename="test.dxf", num=100)`

Quote commands including spaces and always quote the filename argument!

For *xed* on Linux Mint use (note: absolute path to executable):

```
[browse-command]

text_editor = /usr/bin/xed "{filename}" +{num}
icon_size = 32
```

For *gedit* on Linux use (untested):

```
[browse-command]

text_editor = /usr/bin/gedit +{num} "{filename}"
icon_size = 32
```

The *browse* command opens a DXF structure browser to investigate the internals of a DXF file without interpreting the content. The functionality of the DXF browser is similar to the DXF *Pretty Printer* (*pp* command), but without the disadvantage of creating giant HTML files. The intended usage is debugging invalid DXF files, which can not be loaded by the *ezdxf.readfile()* or the *ezdxf.recover.readfile()* functions.

Line Numbers

The low level tag loader ignores DXF comments (group code 999). If there are comments in the DXF file the line numbers displayed in the DXF browser are not synchronized, use the *strip* command beforehand to remove all comments from the DXF file in order to keep the line numbers synchronized.

GUI Features

The tree view on the left shows the outline of the DXF file. The number in round brackets on the right side of each item shows the count of structure entities within the structure layer, the value in angle brackets on the left side is the entity handle.

The right list view shows the entity content as DXF tags. Structure tags (data type <ctrl>) are shown in blue, a double click on a reference handle (datatype <ref>) jumps to the referenced entity, reference handles of non-existent targets are shown in red.

Clicking on the first structure tag in the list opens the DXF reference provided by Autodesk in the standard web browser.

Auto Reload

The browser automatically displays a dialog for reloading DXF files if they have been modified by an external application.

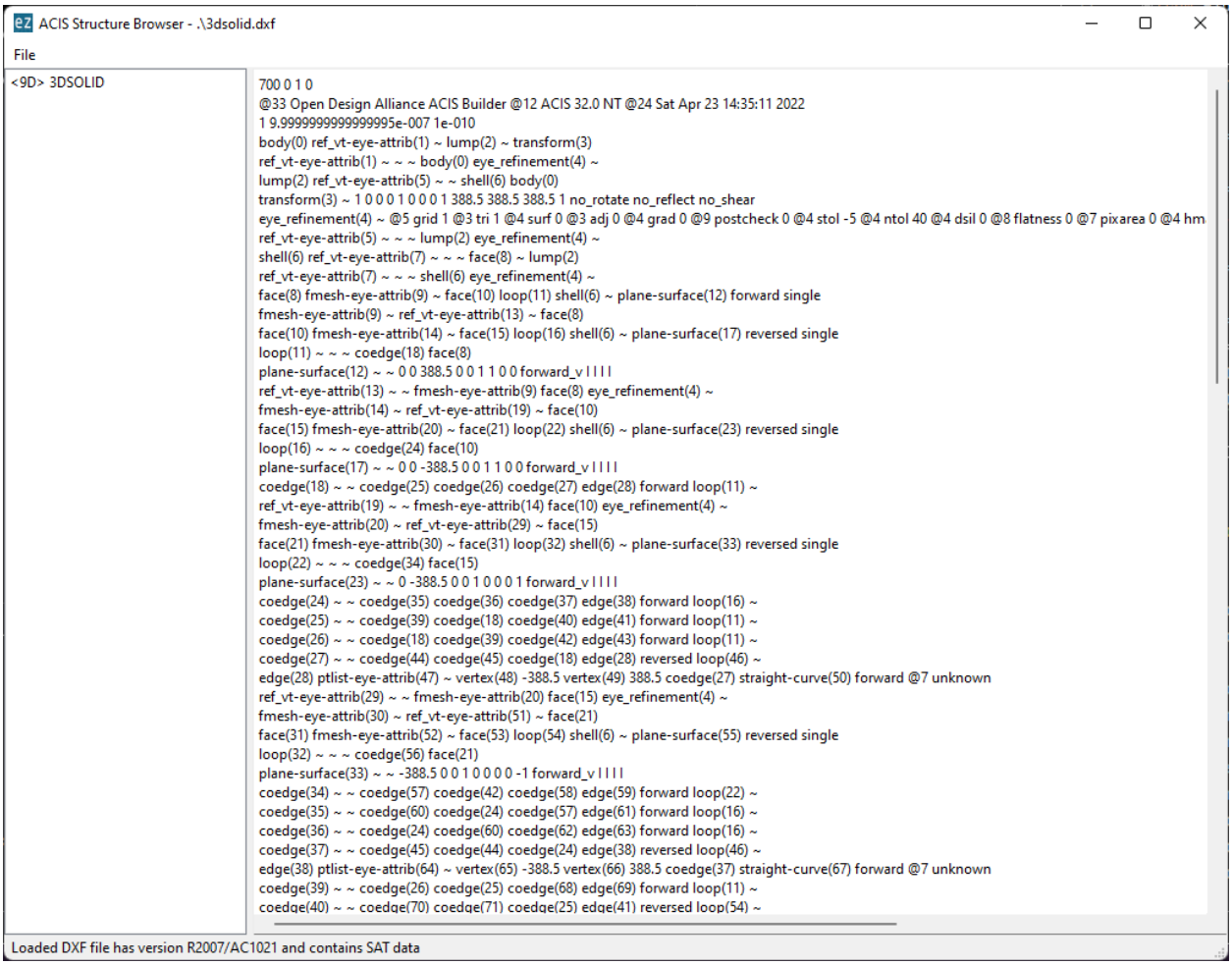
Menus and Shortcuts

- **File Menu**
  - **Open DXF file…** *Ctrl+O*
  - **Reload DXF file** *Ctrl+R*
  - **Open in Text Editor** *Ctrl+T*, open the DXF file in the associated text editor at the current location
  - **Export DXF Entity…** *Ctrl+E*, export the current DXF entity shown in the list view as text file
  - **Copy selected DXF Tags to Clipboard** *Ctrl+C*, copy the current selected DXF tags into the clipboard
  - **Copy DXF Entity to Clipboard** *Ctrl+Shift+C*, copy all DXF tags of the current DXF entity shown in the list view into the clipboard
  - **Quit** *Ctrl+Q*
- **Navigate Menu**
  - **Go to Handle…** *Ctrl+G*
  - **Go to Line…** *Ctrl+L*
  - **Find Text…** *Ctrl+F*, opens the find text dialog
  - **Next Entity** *Ctrl+Right*, go to the next entity in the DXF structure
  - **Previous Entity** *Ctrl+Right*, go to the previous entity in the DXF structure
  - **Show Entity in TreeView** *Ctrl+Down*, expand the left tree view to the currently displayed entity in the list view - this does not happen automatically for performance reasons
  - **Entity History Back** *Alt+Left*
  - **Entity History Forward** *Alt+Right*
  - **Go to HEADERS Section** *Shift+H*
  - **Go to BLOCKS Section** *Shift+B*
  - **Go to ENTITIES Section** *Shift+E*
  - **Go to OBJECTS Section** *Shift+O*
- **Bookmarks Menu**
  - **Store Bookmark…** *Ctrl+Shift+B*, store current location as named bookmark
  - **Go to Bookmark…** *Ctrl+B*, go to stored location

6.10.7 Browse-ACIS

Show and export the *SAT* or *SAB* content of *ACIS* entities:

```
C:\> ezdxf browse-acis 3dsolid.dxf
```



The DXF format stores modern solid geometry as *SAT* data for DXF R2000 - R2010 and as *SAB* data for DXF R2013 and later. This command shows the content of this entities and also let you export the raw data for further processing.

Entity View

The entity view is a read-only text editor, it's possible to select and copy parts of the text into the clipboard. To improve the readability all ACIS entities get automatically an id because AutoCAD and BricsCAD use relative references for ACIS data export and do not assign entity ids. The id is shown as decimal number in parenthesis after the entity name. The ~ character is a shortcut for a null-pointer.

```
C:\>ezdxf browse-acis -h
usage: ezdxf browse-acis [-h] [-g HANDLE] [FILE]

positional arguments:
  FILE                  DXF file to browse
```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  -g HANDLE, --handle HANDLE
                        go to entity by HANDLE, HANDLE has to be a hex value
                        without any prefix like 'fefe'
```

Menus and Shortcuts

- **File Menu**
  - **Open DXF file…** *Ctrl+O*
  - **Reload DXF file** *Ctrl+R*
  - **Export Current Entity View…** *Ctrl+E*, Export the parsed content of the entity view as text file
  - **Export Raw SAT/SAB Data…** *Ctrl+W*, export the raw SAT data as text file and the raw SAB data as a binary file for further processing
  - **Quit** *Ctrl+Q*

6.10.8 Strip

Strip comment tags (group code 999) from ASCII DXF files and can remove the THUMBNAILIMAGE section. Binary DXF files are not supported.

```
C:\> ezdxf strip -h
usage: ezdxf strip [-h] [-b] [-v] FILE [FILE ...]

positional arguments:
  FILE           DXF file to process, wildcards "*" and "?" are supported

optional arguments:
  -h, --help       show this help message and exit
  -b, --backup     make a backup copy with extension ".bak" from the DXF file,
                   overwrites existing backup files
  -t, --thumbnail  strip THUMBNAILIMAGE section
  -v, --verbose    give more output
```

6.10.9 Config

Manage config files.

```
C:\> ezdxf config -h
usage: ezdxf config [-h] [-p] [-w FILE] [--home] [--reset]

optional arguments:
  -h, --help   show this help message and exit
  -p, --print  print configuration
  -w FILE, --write FILE
               write configuration
  --home       create config file 'ezdxf.ini' in the user home directory
               '~/.config/ezdxf', $XDG_CONFIG_HOME is supported if set
```

(continues on next page)

(continued from previous page)

```
  --reset      factory reset, delete default config files 'ezdxf.ini'
```

6.10.10 Info

Show information and optional stats of DXF files as loaded by *ezdxf*, this may not represent the original content of the file, use the *browse* command to see the original content. The upgrade is necessary for very old DXF versions prior to R12 and for the "special" versions R13 and R14. The *-s* option shows some statistics about the DXF content like entity count or table count. Use the *-v* option show more of everything.

```
C:\> ezdxf info -h
usage: ezdxf info [-h] [-v] [-s] FILE [FILE ...]

positional arguments:
  FILE           DXF file to process, wildcards "*" and "?" are supported

options:
  -h, --help     show this help message and exit
  -v, --verbose  give more output
  -s, --stats    show content stats
```

This is the verbose output for an old DXF R10 file and shows that the loading process created some required structures which do not exist in DXF R10 files, like the BLOCK_RECORD table or the OBJECTS section:

```
C:\> ezdxf info -v -s test_R10.dxf

Filename: "test_R10.dxf"
Loaded content was upgraded from DXF Version AC1006 (R10)
Release: R12
DXF Version: AC1009
Maintenance Version: <undefined>
Codepage: ANSI_1252
Encoding: cp1252
Unit system: Imperial
Modelspace units: Unitless
$LASTSAVEDBY: <undefined>
$HANDSEED: 0
$FINGERPRINTGUID: {9EADDC7C-5982-4C68-B770-8A62378C2B90}
$VERSIONGUID: {49336E63-D99B-45EC-803C-4D2BD03A7DE0}
$USERI1=0
$USERI2=0
$USERI3=0
$USERI4=0
$USERI5=0
$USERR1=0.0
$USERR2=0.0
$USERR3=0.0
$USERR4=0.0
$USERR5=0.0
File was not created by ezdxf >= 0.16.4
File was not written by ezdxf >= 0.16.4
Content stats:
LAYER table entries: 18
  0
```

(continues on next page)

(continued from previous page)

```
  Defpoints
  LYR_00
  LYR_01
  LYR_02
  LYR_03
  LYR_04
  LYR_05
  LYR_06
  LYR_07
  LYR_08
  LYR_09
  LYR_10
  LYR_11
  LYR_12
  LYR_13
  LYR_14
  LYR_15
LTYPE table entries: 13
  BORDER
  ByBlock
  ByLayer
  CENTER
  CONTINUOUS
  CUTTING
  DASHDOT
  DASHED
  DIVIDE
  DOT
  HIDDEN
  PHANTOM
  STITCH
STYLE table entries: 1
  STANDARD
DIMSTYLE table entries: 1
  Standard
APPID table entries: 1
  ACAD
UCS table entries: 0
VIEW table entries: 0
VPORT table entries: 1
  *Active
BLOCK_RECORD table entries: 2
  *Model_Space
  *Paper_Space
Entities in modelspace: 78
  ARC (2)
  CIRCLE (2)
  LINE (74)
Entities in OBJECTS section: 20
  ACDBDICTIONARYWDFLT (1)
  ACDBPLACEHOLDER (1)
  DICTIONARY (11)
  LAYOUT (2)
  MATERIAL (3)
  MLEADERSTYLE (1)
  MLINESTYLE (1)
```

6.10.11 Show Version & Configuration

Show the *ezdxf* version and configuration:

```
C:\> ezdxf -Vv

ezdxf v0.16.5b0 @ d:\source\ezdxf.git\src\ezdxf
Python version: 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit␣
↪(AMD64)]
using C-extensions: yes
using Matplotlib: yes

Configuration:
[core]
default_dimension_text_style = OpenSansCondensed-Light
test_files = D:\Source\dxftest
font_cache_directory =
load_proxy_graphics = true
store_proxy_graphics = true
log_unprocessed_tags = false
filter_invalid_xdata_group_codes = true
write_fixed_meta_data_for_testing = false
disable_c_ext = false

[browse-command]
text_editor = "C:\Program Files\Notepad++\notepad++.exe" "{filename}" -n{num}


Environment Variables:
EZDXF_DISABLE_C_EXT=
EZDXF_TEST_FILES=D:\Source\dxftest
EZDXF_CONFIG_FILE=

Existing Configuration Files:
C:\Users\manfred\.config\ezdxf\ezdxf.ini
```

See also:

Documentation of the `ezdxf.options` module and the *Environment Variables*.

6.11 Rendering

The `ezdxf.render` subpackage provides helpful utilities to create complex forms.

- create complex meshes as `Mesh` entity.
- render complex curves like bezier curves, euler spirals or splines as `Polyline` entity
- vertex generators for simple and complex forms like circle, ellipse or euler spiral

Content

6.11.1 Spline

class ezdxf.render.**Spline**(*points: Iterable[*UVec*] | None = None*, *segments: int = 100*)

This class can be used to render B-splines into DXF R12 files as approximated *Polyline* entities. The advantage of this class over the *R12Spline* class is, that this is a real 3D curve, which means that the B-spline vertices do have to be located in a flat plane, and no *UCS* class is needed to place the curve in 3D space.

See also:

The newer *BSpline* class provides the advanced vertex interpolation method *flattening()*.

__init__(*points: Iterable[*UVec*] | None = None*, *segments: int = 100*)

Parameters

- **points** – spline definition points
- **segments** – count of line segments for approximation, vertex count is *segments* + 1

subdivide(*segments: int = 4*) → None

Calculate overall segment count, where segments is the sub-segment count, *segments* = 4, means 4 line segments between two definition points e.g. 4 definition points and 4 segments = 12 overall segments, useful for fit point rendering.

Parameters

segments – sub-segments count between two definition points

render_as_fit_points(*layout:* BaseLayout, *degree: int = 3*, *method: str = 'chord'*, *dxfattribs: dict | None = None*) → None

Render a B-spline as 2D/3D *Polyline*, where the definition points are fit points.

- 2D spline vertices uses: *add_polyline2d()*
- 3D spline vertices uses: *add_polyline3d()*

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **method** – “uniform”, “distance”/”chord”, “centripetal”/”sqrt_chord” or “arc” calculation method for parameter t
- **dxfattribs** – DXF attributes for *Polyline*

render_open_bspline(*layout:* BaseLayout, *degree: int = 3*, *dxfattribs=None*) → None

Render an open uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_uniform_bspline(*layout: BaseLayout*, *degree: int = 3*, *dxfattribs=None*) → None

Render a uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_closed_bspline(*layout: BaseLayout*, *degree: int = 3*, *dxfattribs=None*) → None

Render a closed uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_open_rbspline(*layout: BaseLayout*, *weights: Iterable[float]*, *degree: int = 3*, *dxfattribs=None*) → None

Render a rational open uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **weights** – list of weights, requires a weight value (float) for each definition point.
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_uniform_rbspline(*layout: BaseLayout*, *weights: Iterable[float]*, *degree: int = 3*, *dxfattribs=None*) → None

Render a rational uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **weights** – list of weights, requires a weight value (float) for each definition point.
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_closed_rbspline(*layout: BaseLayout*, *weights: Iterable[float]*, *degree: int = 3*, *dxfattribs=None*) → None

Render a rational B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **weights** – list of weights, requires a weight value (float) for each definition point.
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

6.11.2 R12Spline

class ezdxf.render.**R12Spline**(*control_points: Iterable[*UVec*]*, *degree: int = 2*, *closed: bool = True*)

DXF R12 supports 2D B-splines, but Autodesk do not document the usage in the DXF Reference. The base entity for splines in DXF R12 is the POLYLINE entity. The spline itself is always in a plane, but as any 2D entity, the spline can be transformed into the 3D object by elevation and extrusion (*OCS*, *UCS*).

This way it was possible to store the spline parameters in the DXF R12 file, to allow CAD applications to modify the spline parameters and rerender the B-spline afterward again as polyline approximation. Therefore, the result is not better than an approximation by the `Spline` class, it is also just a POLYLINE entity, but maybe someone need exact this tool in the future.

__init__(*control_points: Iterable[*UVec*]*, *degree: int = 2*, *closed: bool = True*)

Parameters

- **control_points** – B-spline control frame vertices
- **degree** – degree of B-spline, only 2 and 3 is supported
- **closed** – `True` for closed curve

render(*layout:* BaseLayout, *segments: int = 40*, *ucs:* UCS | *None = None*, *dxfattribs=None*) → *Polyline*

Renders the B-spline into *layout* as 2D `Polyline` entity. Use an `UCS` to place the 2D spline in the 3D space, see `approximate()` for more information.

Parameters

- **layout** – `BaseLayout` object
- **segments** – count of line segments for approximation, vertex count is *segments* + 1
- **ucs** – `UCS` definition, control points in ucs coordinates.
- **dxfattribs** – DXF attributes for `Polyline`

approximate(*segments: int = 40*, *ucs:* UCS | *None = None*) → list[*UVec*]

Approximate the B-spline by a polyline with *segments* line segments. If *ucs* is not `None`, ucs defines an `UCS`, to transform the curve into *OCS*. The control points are placed xy-plane of the UCS, don't use z-axis coordinates, if so make sure all control points are in a plane parallel to the OCS base plane (UCS xy-plane), else the result is unpredictable and depends on the CAD application used to open the DXF file - it may crash.

Parameters

- **segments** – count of line segments for approximation, vertex count is *segments* + 1
- **ucs** – `UCS` definition, control points in ucs coordinates

Returns

list of vertices in `OCS` as `Vec3` objects

6.11.3 Bezier

class ezdxf.render.**Bezier**

Render a bezier curve as 2D/3D *Polyline*.

The *Bezier* class is implemented with multiple segments, each segment is an optimized 4 point bezier curve, the 4 control points of the curve are: the start point (1) and the end point (4), point (2) is start point + start vector and point (3) is end point + end vector. Each segment has its own approximation count.

See also:

The new *ezdxf.path* package provides many advanced construction tools based on the *Path* class.

start (*point:* UVec, *tangent:* UVec) → None

Set start point and start tangent.

Parameters

- **point** – start point
- **tangent** – start tangent as vector, example: (5, 0, 0) means a horizontal tangent with a length of 5 drawing units

append (*point:* UVec, *tangent1:* UVec, *tangent2:* UVec | *None* = *None*, *segments: int* = *20*)

Append a control point with two control tangents.

Parameters

- **point** – control point
- **tangent1** – first tangent as vector "left" of the control point
- **tangent2** – second tangent as vector "right" of the control point, if omitted *tangent2* = *-tangent1*
- **segments** – count of line segments for the polyline approximation, count of line segments from the previous control point to the appended control point.

render (*layout:* BaseLayout, *force3d: bool* = *False*, *dxfattribs*=*None*) → None

Render Bezier curve as 2D/3D *Polyline*.

Parameters

- **layout** – *BaseLayout* object
- **force3d** – force 3D polyline rendering
- **dxfattribs** – DXF attributes for *Polyline*

6.11.4 EulerSpiral

class ezdxf.render.**EulerSpiral**(*curvature: float* = *1*)

Render an euler spiral as a 3D *Polyline* or a *Spline* entity.

This is a parametric curve, which always starts at the origin (0, 0).

__init__(*curvature: float* = *1*)

Parameters

curvature – Radius of curvature

render_polyline(*layout: BaseLayout*, *length: float = 1*, *segments: int = 100*, *matrix: Matrix44 | None = None*, *dxfattribs=None*)

Render curve as *Polyline*.

Parameters

- **layout** – *BaseLayout* object
- **length** – length measured along the spiral curve from its initial position
- **segments** – count of line segments to use, vertex count is *segments* + 1
- **matrix** – transformation matrix as *Matrix44*
- **dxfattribs** – DXF attributes for *Polyline*

Returns
Polyline

render_spline(*layout: BaseLayout*, *length: float = 1*, *fit_points: int = 10*, *degree: int = 3*, *matrix: Matrix44 | None = None*, *dxfattribs=None*)

Render curve as *Spline*.

Parameters

- **layout** – *BaseLayout* object
- **length** – length measured along the spiral curve from its initial position
- **fit_points** – count of spline fit points to use
- **degree** – degree of B-spline
- **matrix** – transformation matrix as *Matrix44*
- **dxfattribs** – DXF attributes for *Spline*

Returns
Spline

6.11.5 Random Paths

Random path generators for testing purpose.

ezdxf.render.**random_2d_path**(*steps: int = 100*, *max_step_size: float = 1.0*, *max_heading: float = math.pi / 2*, *retarget: int = 20*) → Iterable[*Vec2*]

Returns a random 2D path as iterable of *Vec2* objects.

Parameters

- **steps** – count of vertices to generate
- **max_step_size** – max step size
- **max_heading** – limit heading angle change per step to ± max_heading/2 in radians
- **retarget** – specifies steps before changing global walking target

ezdxf.render.**random_3d_path**(*steps: int = 100*, *max_step_size: float = 1.0*, *max_heading: float = math.pi / 2.0*, *max_pitch: float = math.pi / 8.0*, *retarget: int = 20*) → Iterable[*Vec3*]

Returns a random 3D path as iterable of *Vec3* objects.

Parameters

- **steps** – count of vertices to generate
- **max_step_size** – max step size
- **max_heading** – limit heading angle change per step to ± max_heading/2, rotation about the z-axis in radians
- **max_pitch** – limit pitch angle change per step to ± max_pitch/2, rotation about the x-axis in radians
- **retarget** – specifies steps before changing global walking target

6.11.6 Forms

This module provides functions to create 2D and 3D forms as vertices or mesh objects.

2D Forms

- *box()*
- *circle()*
- *ellipse()*
- *euler_spiral()*
- *gear()*
- *ngon()*
- *square()*
- *star()*
- *turtle()*

3D Forms

- *cone_2p()*
- *cone()*
- *cube()*
- *cylinder()*
- *cylinder_2p()*
- *helix()*
- *sphere()*
- *torus()*

3D Form Builder

- *extrude()*
- *extrude_twist_scale()*
- *from_profiles_linear()*
- *from_profiles_spline()*
- *rotation_form()*
- *sweep()*

- *sweep_profile()*

2D Forms

Basic 2D shapes as iterable of *Vec3*.

ezdxf.render.forms.**box**(*sx: float = 1.0*, *sy: float = 1.0*, *center=False*) → tuple[*ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*]

Returns 4 vertices for a box with a width of *sx* by and a height of *sy*. The center of the box in (0, 0) if *center* is `True` otherwise the lower left corner is (0, 0), upper right corner is (*sx*, *sy*).

ezdxf.render.forms.**circle**(*count: int*, *radius: float = 1*, *elevation: float = 0*, *close: bool = False*) → Iterable[*Vec3*]

Create polygon vertices for a circle with the given *radius* and approximated by *count* vertices, *elevation* is the z-axis for all vertices.

Parameters

- **count** – count of polygon vertices
- **radius** – circle radius
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if `True`.

Returns
vertices in counter-clockwise orientation as *Vec3* objects

ezdxf.render.forms.**ellipse**(*count: int*, *rx: float = 1*, *ry: float = 1*, *start_param: float = 0*, *end_param: float = math.tau*, *elevation: float = 0*) → Iterable[*Vec3*]

Create polygon vertices for an ellipse with given *rx* as x-axis radius and *ry* as y-axis radius approximated by *count* vertices, *elevation* is the z-axis for all vertices. The ellipse goes from *start_param* to *end_param* in counter clockwise orientation.

Parameters

- **count** – count of polygon vertices
- **rx** – ellipse x-axis radius
- **ry** – ellipse y-axis radius
- **start_param** – start of ellipse in range [0, 2π]
- **end_param** – end of ellipse in range [0, 2π]
- **elevation** – z-axis for all vertices

Returns
vertices in counter clockwise orientation as *Vec3* objects

ezdxf.render.forms.**euler_spiral**(*count: int*, *length: float = 1*, *curvature: float = 1*, *elevation: float = 0*) → Iterable[*Vec3*]

Create polygon vertices for an euler spiral of a given *length* and radius of curvature. This is a parametric curve, which always starts at the origin (0, 0).

Parameters

- **count** – count of polygon vertices
- **length** – length of curve in drawing units

- **curvature** – radius of curvature
- **elevation** – z-axis for all vertices

Returns
vertices as *Vec3* objects

ezdxf.render.forms.**gear**(*count: int*, *top_width: float*, *bottom_width: float*, *height: float*, *outside_radius: float*, *elevation: float = 0*, *close: bool = False*) → Iterable[*Vec3*]

Returns the corner vertices of a gear shape (cogwheel).

Warning: This function does not create correct gears for mechanical engineering!

Parameters

- **count** – teeth count >= 3
- **top_width** – teeth width at outside radius
- **bottom_width** – teeth width at base radius
- **height** – teeth height; base radius = outside radius - height
- **outside_radius** – outside radius
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if True.

Returns
vertices in counter clockwise orientation as *Vec3* objects

ezdxf.render.forms.**ngon**(*count: int*, *length: float | None = None*, *radius: float | None = None*, *rotation: float = 0.0*, *elevation: float = 0.0*, *close: bool = False*) → Iterable[*Vec3*]

Returns the corner vertices of a regular polygon. The polygon size is determined by the edge *length* or the circum *radius* argument. If both are given *length* has the higher priority.

Parameters

- **count** – count of polygon corners >= 3
- **length** – length of polygon side
- **radius** – circum radius
- **rotation** – rotation angle in radians
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if `True`.

Returns
vertices as *Vec3* objects

ezdxf.render.forms.**square**(*size: float = 1.0*, *center=False*) → tuple[*ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*, *ezdxf.math._vector.Vec3*]

Returns 4 vertices for a square with a side length of the given *size*. The center of the square in (0, 0) if *center* is `True` otherwise the lower left corner is (0, 0), upper right corner is (*size*, *size*).

ezdxf.render.forms.**star**(*count: int*, *r1: float*, *r2: float*, *rotation: float = 0.0*, *elevation: float = 0.0*, *close: bool = False*) → Iterable[*Vec3*]

Returns the corner vertices for a star shape.

The shape has *count* spikes, *r1* defines the radius of the "outer" vertices and *r2* defines the radius of the "inner" vertices, but this does not mean that *r1* has to be greater than *r2*.

Parameters

- **count** – spike count >= 3
- **r1** – radius 1
- **r2** – radius 2
- **rotation** – rotation angle in radians
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if `True`.

Returns
vertices as *Vec3* objects

ezdxf.render.forms.**turtle**(*commands: str*, *start=Vec2(0, 0)*, *angle: float = 0*) → Iterator[*Vec2*]

Returns the 2D vertices of a polyline created by turtle-graphic like commands:

- `<length>` - go <length> units forward in current direction and yield vertex
- `r<angle>` - turn right <angle> in degrees, a missing angle is 90 deg
- `l<angle>` - turn left <angle> in degrees, a missing angle is 90 deg
- `@<x>,<y>` - go relative <x>,<y> and yield vertex

The command string `"10 l 10 l 10"` returns the 4 corner vertices of a square with a side length of 10 drawing units.

Parameters

- **commands** – command string, commands are separated by spaces
- **start** – starting point, default is (0, 0)
- **angle** – starting direction, default is 0 deg

3D Forms

Create 3D forms as *MeshTransformer* objects.

ezdxf.render.forms.**cube**(*center: bool = True*) → MeshTransformer

Create a cube as *MeshTransformer* object.

Parameters
center – 'mass' center of cube, `(0, 0, 0)` if `True`, else first corner at `(0, 0, 0)`

Returns: *MeshTransformer*

ezdxf.render.forms.**cone**(*count: int = 16*, *radius: float = 1.0*, *apex: UVec = (0, 0, 1)*, ***, *caps=True*) → MeshTransformer

Create a cone as *MeshTransformer* object, the base center is fixed in the origin (0, 0, 0).

Parameters

- **count** – edge count of basis_vector

- **radius** – radius of basis_vector
- **apex** – tip of the cone
- **caps** – add a bottom face as ngon if `True`

ezdxf.render.forms.**cone_2p**(*count: int = 16*, *radius: float = 1.0*, *base_center:* UVec *= (0, 0, 0)*, *apex:* UVec *= (0, 0, 1)*, *, *caps=True*) → MeshTransformer

Create a cone as *MeshTransformer* object from two points, *base_center* is the center of the base circle and *apex* as the tip of the cone.

Parameters

- **count** – edge count of basis_vector
- **radius** – radius of basis_vector
- **base_center** – center point of base circle
- **apex** – tip of the cone
- **caps** – add a bottom face as ngon if `True`

Raises

ValueError – the cone orientation cannot be detected (base center == apex)

ezdxf.render.forms.**cylinder**(*count: int = 16*, *radius: float = 1.0*, *top_radius: float | None = None*, *top_center:* UVec *= (0, 0, 1)*, *, *caps=True*) → MeshTransformer

Create a cylinder as *MeshTransformer* object, the base center is fixed in the origin (0, 0, 0).

Parameters

- **count** – profiles edge count
- **radius** – radius for bottom profile
- **top_radius** – radius for top profile, if `None` top_radius == radius
- **top_center** – location vector for the center of the top profile
- **caps** – close hull with top- and bottom faces (ngons)

ezdxf.render.forms.**cylinder_2p**(*count: int = 16*, *radius: float = 1*, *base_center:* UVec *= (0, 0, 0)*, *top_center:* UVec *= (0, 0, 1)*, *, *caps=True*) → MeshTransformer

Creates a cylinder as *MeshTransformer* object from two points, *base_center* is the center of the base circle and, *top_center* the center of the top circle.

Parameters

- **count** – cylinder profile edge count
- **radius** – radius for bottom profile
- **base_center** – center of base circle
- **top_center** – center of top circle
- **caps** – close hull with top- and bottom faces (ngons)

Raises

ValueError – the cylinder orientation cannot be detected (base center == top center)

ezdxf.render.forms.**helix**(*radius: float*, *pitch: float*, *turns: float*, *resolution: int = 16*, *ccw=True*) → Iterator[*Vec3*]

Yields the vertices of a helix. The center of the helix is always (0, 0), a positive *pitch* value creates a helix along the +z-axis, a negative value along the -z-axis.

Parameters

- **radius** – helix radius
- **pitch** – the height of one complete helix turn
- **turns** – count of turns
- **resolution** – vertices per turn
- **ccw** – creates a counter-clockwise turning (right-handed) helix if `True`

`ezdxf.render.forms.`**sphere**(*count: int = 16*, *stacks: int = 8*, *radius: float = 1*, ***, *quads=True*) → MeshTransformer

Create a sphere as *MeshTransformer* object, the center of the sphere is always at (0, 0, 0).

Parameters

- **count** – longitudinal slices
- **stacks** – latitude slices
- **radius** – radius of sphere
- **quads** – use quadrilaterals as faces if `True` else triangles

`ezdxf.render.forms.`**torus**(*major_count: int = 16*, *minor_count: int = 8*, *major_radius=1.0*, *minor_radius=0.1*, *start_angle: float = 0.0*, *end_angle: float = math.tau*, ***, *caps=True*) → MeshTransformer

Create a torus as *MeshTransformer* object, the center of the torus is always at (0, 0, 0). The *major_radius* has to be bigger than the *minor_radius*.

Parameters

- **major_count** – count of circles
- **minor_count** – count of circle vertices
- **major_radius** – radius of the circle center
- **minor_radius** – radius of circle
- **start_angle** – start angle of torus in radians
- **end_angle** – end angle of torus in radians
- **caps** – close hull with start- and end faces (ngons) if the torus is open

3D Form Builder

`ezdxf.render.forms.`**extrude**(*profile: Iterable[*UVec*]*, *path: Iterable[*UVec*]*, *close=True*, *caps=False*) → MeshTransformer

Extrude a *profile* polygon along a *path* polyline, the vertices of *profile* should be in counter-clockwise order. The sweeping profile will not be rotated at extrusion!

Parameters

- **profile** – sweeping profile as list of (x, y, z) tuples in counter-clockwise order
- **path** – extrusion path as list of (x, y, z) tuples
- **close** – close profile polygon if `True`
- **caps** – close hull with top- and bottom faces (ngons)

Returns: *MeshTransformer*

ezdxf.render.forms.**extrude_twist_scale**(*profile: Iterable[UVec]*, *path: Iterable[UVec]*, ***, *twist: float = 0.0*, *scale: float = 1.0*, *step_size: float = 1.0*, *close=True*, *caps=False*, *quads=True*) → MeshTransformer

Extrude a *profile* polygon along a *path* polyline, the vertices of *profile* should be in counter-clockwise order. This implementation can scale and twist the sweeping profile along the extrusion path. The *path* segment points are fix points, the *max_step_size* is used to create intermediate profiles between this fix points. The *max_step_size* is adapted for each segment to create equally spaced distances. The twist angle is the rotation angle in radians and the scale *argument* defines the scale factor of the final profile. The twist angle and scaling factor of the intermediate profiles will be linear interpolated between the start and end values.

Parameters

- **profile** – sweeping profile as list of (x, y, z) tuples in counter-clockwise order
- **path** – extrusion path as list of (x, y, z) tuples
- **twist** – rotate sweeping profile up to the given end rotation angle in radians
- **scale** – scale sweeping profile gradually from 1.0 to given value
- **step_size** – rough distance between automatically created intermediate profiles, the step size is adapted to the distances between the path segment points, a value od 0.0 disables creating intermediate profiles
- **close** – close profile polygon if `True`
- **caps** – close hull with top- and bottom faces (ngons)
- **quads** – use quads for "sweeping" faces if `True` else triangles, the top and bottom faces are always ngons

Returns: *MeshTransformer*

ezdxf.render.forms.**from_profiles_linear**(*profiles: Sequence[Sequence[Vec3]]*, ***, *close=True*, *quads=True*, *caps=False*) → MeshTransformer

Returns a *MeshTransformer* instance from linear connected *profiles*.

Parameters

- **profiles** – list of profiles
- **close** – close profile polygon if `True`
- **quads** – use quadrilaterals as connection faces if `True` else triangles
- **caps** – close hull with top- and bottom faces (ngons)

ezdxf.render.forms.**from_profiles_spline**(*profiles: Sequence[Sequence[Vec3]]*, *subdivide: int = 4*, ***, *close=True*, *quads=True*, *caps=False*) → MeshTransformer

Returns a *MeshTransformer* instance by spline interpolation between given *profiles*. Requires at least 4 profiles. A *subdivide* value of 4, means, create 4 face loops between two profiles, without interpolation two profiles create one face loop.

Parameters

- **profiles** – list of profiles
- **subdivide** – count of face loops
- **close** – close profile polygon if `True`

- **quads** – use quadrilaterals as connection faces if `True` else triangles
- **caps** – close hull with top- and bottom faces (ngons)

ezdxf.render.forms.**rotation_form**(*count: int*, *profile: Iterable[*UVec*]*, *angle: float = math.tau*, *axis:* UVec *= (1, 0, 0)*, ***, *caps=False*) → MeshTransformer

Returns a *MeshTransformer* instance created by rotating a *profile* around an *axis*.

Parameters

- **count** – count of rotated profiles
- **profile** – profile to rotate as list of vertices
- **angle** – rotation angle in radians
- **axis** – rotation axis
- **caps** – close hull with start- and end faces (ngons)

ezdxf.render.forms.**sweep**(*profile: Iterable[*UVec*]*, *sweeping_path: Iterable[*UVec*]*, ***, *close=True*, *quads=True*, *caps=True*) → MeshTransformer

Returns the mesh from sweeping a profile along a 3D path, where the sweeping path defines the final location in the *WCS*.

The profile is defined in a reference system. The origin of this reference system will be moved along the sweeping path where the z-axis of the reference system is pointing into the moving direction.

Returns the mesh as *ezdxf.render.MeshTransformer* object.

Parameters

- **profile** – sweeping profile defined in the reference system as iterable of (x, y, z) coordinates in counter-clockwise order
- **sweeping_path** – the sweeping path defined in the WCS as iterable of (x, y, z) coordinates
- **close** – close sweeping profile if `True`
- **quads** – use quadrilaterals as connection faces if `True` else triangles
- **caps** – close hull with top- and bottom faces (ngons)

ezdxf.render.forms.**sweep_profile**(*profile: Iterable[*UVec*]*, *sweeping_path: Iterable[*UVec*]*) → list[Sequence[*ezdxf.math._vector.Vec3*]]

Returns the intermediate profiles of sweeping a profile along a 3D path where the sweeping path defines the final location in the *WCS*.

The profile is defined in a reference system. The origin of this reference system will be moved along the sweeping path where the z-axis of the reference system is pointing into the moving direction.

Returns the start-, end- and all intermediate profiles along the sweeping path.

6.11.7 MeshBuilder

The *MeshBuilder* classes are helper tools to manage meshes buildup by vertices and faces. The vertices are stored in a vertices list as `Vec3` instances. The faces are stored as a sequence of vertex indices which is the location of the vertex in the vertex list. A single *MeshBuilder* class can contain multiple separated meshes at the same time.

The method *MeshBuilder.render_mesh()* renders the content as a single DXF *Mesh* entity, which supports ngons, ngons are polygons with more than 4 vertices. This entity requires at least DXF R2000.

The method *MeshBuilder.render_polyface()* renders the content as a single DXF *Polyface* entity, which supports only triangles and quadrilaterals. This entity is supported by DXF R12.

The method *MeshBuilder.render_3dfaces()* renders each face of the mesh as a single DXF *Face3d* entity, which supports only triangles and quadrilaterals. This entity is supported by DXF R12.

The *MeshTransformer* class is often used as an interface object to transfer mesh data between functions and moduls, like for the mesh exchange add-on *meshex*.

The basic *MeshBuilder* class does not support transformations.

class ezdxf.render.**MeshBuilder**

vertices

List of vertices as *Vec3* or `(x, y, z)` tuple

faces

List of faces as list of vertex indices, where a vertex index is the index of the vertex in the *vertices* list. A face requires at least three vertices, *Mesh* supports ngons, so the count of vertices is not limited.

add_face(*vertices: Iterable[*UVec*]*) → None

Add a face as vertices list to the mesh. A face requires at least 3 vertices, each vertex is a `(x, y, z)` tuple or *Vec3* object. The new vertex indices are stored as face in the *faces* list.

Parameters

vertices – list of at least 3 vertices `[(x1, y1, z1), (x2, y2, z2), (x3, y3, y3), ...]`

add_mesh(*vertices: list[*ezdxf.math._vector.Vec3*]* | *None* = *None*, *faces: list[Sequence[int]]* | *None* = *None*, *mesh=None*) → None

Add another mesh to this mesh.

A *mesh* can be a *MeshBuilder*, *MeshVertexMerger* or *Mesh* object or requires the attributes *vertices* and *faces*.

Parameters

- **vertices** – list of vertices, a vertex is a `(x, y, z)` tuple or *Vec3* object
- **faces** – list of faces, a face is a list of vertex indices
- **mesh** – another mesh entity

add_vertices(*vertices: Iterable[*UVec*]*) → Sequence[int]

Add new vertices to the mesh, each vertex is a `(x, y, z)` tuple or a *Vec3* object, returns the indices of the *vertices* added to the *vertices* list.

e.g. adding 4 vertices to an empty mesh, returns the indices `(0, 1, 2, 3)`, adding additional 4 vertices returns the indices `(4, 5, 6, 7)`.

Parameters

vertices – list of vertices, vertex as `(x, y, z)` tuple or *Vec3* objects

Returns
indices of the *vertices* added to the `vertices` list

Return type
tuple

bbox() → *BoundingBox*

Returns the `BoundingBox` of the mesh.

copy()

Returns a copy of mesh.

diagnose() → MeshDiagnose

Returns the `MeshDiagnose` object for this mesh.

face_normals() → Iterator[*Vec3*]

Yields all face normals, yields the `NULLVEC` instance for degenerated faces.

face_orientation_detector(*reference: int = 0*) → *FaceOrientationDetector*

Returns a `FaceOrientationDetector` or short *fod* instance. The forward orientation is defined by the *reference* face which is 0 by default.

The *fod* can check if all faces are reachable from the reference face and if all faces have the same orientation. The *fod* can be reused to unify the face orientation of the mesh.

faces_as_vertices() → Iterator[list[*ezdxf.math._vector.Vec3*]]

Yields all faces as list of vertices.

flip_normals() → None

Flips the normals of all faces by reversing the vertex order inplace.

classmethod from_builder(*other:* MeshBuilder)

Create new mesh from other mesh builder, faster than `from_mesh()` but supports only `MeshBuilder` and inherited classes.

classmethod from_mesh(*other:* MeshBuilder | Mesh) → T

Create new mesh from other mesh as class method.

Parameters
other – *mesh* of type `MeshBuilder` and inherited or DXF `Mesh` entity or any object providing attributes `vertices`, `edges` and `faces`.

classmethod from_polyface(*other:* Polymesh | Polyface) → T

Create new mesh from a `Polyface` or `Polymesh` object.

get_face_vertices(*index: int*) → Sequence[*Vec3*]

Returns the face *index* as sequence of `Vec3` objects.

get_face_normal(*index: int*) → *Vec3*

Returns the normal vector of the face *index* as `Vec3`, returns the `NULLVEC` instance for degenerated faces.

merge_coplanar_faces(*passes: int = 1*) → MeshTransformer

Returns a new `MeshBuilder` object with merged adjacent coplanar faces.

The faces have to share at least two vertices and have to have the same clockwise or counter-clockwise vertex order.

The current implementation is not very capable!

mesh_tessellation(*max_vertex_count: int = 4*) → MeshTransformer

Returns a new *MeshTransformer* instance, where each face has no more vertices than the given *max_vertex_count*.

The *fast* mode uses a shortcut for faces with less than 6 vertices which may not work for concave faces!

normalize_faces() → None

Removes duplicated vertex indices from faces and stores all faces as open faces, where the last vertex is not coincident with the first vertex.

open_faces() → Iterator[Sequence[int]]

Yields all faces as sequence of integers where the first vertex is not coincident with the last vertex.

optimize_vertices(*precision: int = 6*) → MeshTransformer

Returns a new mesh with optimized vertices. Coincident vertices are merged together and all faces are open faces (first vertex != last vertex). Uses internally the *MeshVertexMerger* class to merge vertices.

render_3dfaces(*layout: GenericLayoutType*, *dxfattribs=None*, *matrix:* Matrix44 | *None = None*, *ucs:* UCS | *None = None*)

Render mesh as *Face3d* entities into *layout*.

Parameters

- **layout** – *BaseLayout* object
- **dxfattribs** – dict of DXF attributes e.g. `{'layer': 'mesh', 'color': 7}`
- **matrix** – transformation matrix of type *Matrix44*
- **ucs** – transform vertices by *UCS* to *WCS*

render_mesh(*layout: GenericLayoutType*, *dxfattribs=None*, *matrix:* Matrix44 | *None = None*, *ucs:* UCS | *None = None*)

Render mesh as *Mesh* entity into *layout*.

Parameters

- **layout** – *BaseLayout* object
- **dxfattribs** – dict of DXF attributes e.g. `{'layer': 'mesh', 'color': 7}`
- **matrix** – transformation matrix of type *Matrix44*
- **ucs** – transform vertices by *UCS* to *WCS*

render_normals(*layout: GenericLayoutType*, *length: float = 1*, *relative=True*, *dxfattribs=None*)

Render face normals as *Line* entities into *layout*, useful to check orientation of mesh faces.

Parameters

- **layout** – *BaseLayout* object
- **length** – visual length of normal, use length < 0 to point normals in opposite direction
- **relative** – scale length relative to face size if `True`
- **dxfattribs** – dict of DXF attributes e.g. `{'layer': 'normals', 'color': 6}`

render_polyface(*layout: GenericLayoutType*, *dxfattribs=None*, *matrix:* Matrix44 | *None = None*, *ucs:* UCS | *None = None*)

Render mesh as *Polyface* entity into *layout*.

Parameters

- **layout** – *BaseLayout* object
- **dxfattribs** – dict of DXF attributes e.g. `{'layer': 'mesh', 'color': 7}`
- **matrix** – transformation matrix of type *Matrix44*
- **ucs** – transform vertices by *UCS* to *WCS*

separate_meshes() → list[ezdxf.render.mesh.MeshTransformer]

A single *MeshBuilder* instance can store multiple separated meshes. This function returns this separated meshes as multiple *MeshTransformer* instances.

subdivide(*level: int = 1*, *quads=True*) → MeshTransformer

Returns a new *MeshTransformer* object with all faces subdivided.

Parameters

- **level** – subdivide levels from 1 to max of 5
- **quads** – create quad faces if `True` else create triangles

subdivide_ngons(*max_vertex_count=4*) → Iterator[Sequence[*Vec3*]]

Yields all faces as sequence of *Vec3* instances, where all ngons which have more than *max_vertex_count* vertices gets subdivided. In contrast to the *tessellation()* method, creates this method a new vertex in the centroid of the face. This can create a more regular tessellation but only works reliable for convex faces!

tessellation(*max_vertex_count: int = 4*) → Iterator[Sequence[*Vec3*]]

Yields all faces as sequence of *Vec3* instances, each face has no more vertices than the given *max_vertex_count*. This method uses the "ear clipping" algorithm which works with concave faces too and does not create any additional vertices.

unify_face_normals(*, *fod:* FaceOrientationDetector | *None = None*) → MeshTransformer

Returns a new *MeshTransformer* object with unified face normal vectors of all faces. The forward direction (not necessarily outwards) is defined by the face-normals of the majority of the faces. This function can not process non-manifold meshes (more than two faces are connected by a single edge) or multiple disconnected meshes in a single *MeshBuilder* object.

It is possible to pass in an existing *FaceOrientationDetector* instance as argument *fod*.

Raises

- **NonManifoldError** – non-manifold mesh
- **MultipleMeshesError** – the *MeshBuilder* object contains multiple disconnected meshes

unify_face_normals_by_reference(*reference: int = 0*, *, *force_outwards=False*, *fod:* FaceOrientationDetector | *None = None*) → MeshTransformer

Returns a new *MeshTransformer* object with unified face normal vectors of all faces. The forward direction (not necessarily outwards) is defined by the reference face, which is the first face of the *mesh* by default. This function can not process non-manifold meshes (more than two faces are connected by a single edge) or multiple disconnected meshes in a single *MeshBuilder* object.

The outward direction of all face normals can be forced by stetting the argument *force_outwards* to `True` but this works only for closed surfaces, and it's time-consuming!

It is not possible to check for a closed surface as long the face normal vectors are not unified. But it can be done afterward by the attribute *MeshDiagnose.is_closed_surface()* to see if the result is trustworthy.

It is possible to pass in an existing *FaceOrientationDetector* instance as argument *fod*.

Parameters

- **reference** – index of the reference face
- **force_outwards** – forces face-normals to point outwards, this works only for closed surfaces, and it's time-consuming!
- **fod** – *FaceOrientationDetector* instance

Raises
ValueError – non-manifold mesh or the *MeshBuilder* object contains multiple disconnected meshes

6.11.8 MeshTransformer

Same functionality as *MeshBuilder* but supports inplace transformation.

class ezdxf.render.**MeshTransformer**

Subclass of *MeshBuilder*

transform(*matrix:* Matrix44)

Transform mesh inplace by applying the transformation *matrix*.

Parameters
matrix – 4x4 transformation matrix as *Matrix44* object

translate(*dx: float* | UVec = *0*, *dy: float = 0*, *dz: float = 0*)

Translate mesh inplace.

Parameters

- **dx** – translation in x-axis or translation vector
- **dy** – translation in y-axis
- **dz** – translation in z-axis

scale(*sx: float = 1*, *sy: float = 1*, *sz: float = 1*)

Scale mesh inplace.

Parameters

- **sx** – scale factor for x-axis
- **sy** – scale factor for y-axis
- **sz** – scale factor for z-axis

scale_uniform(*s: float*)

Scale mesh uniform inplace.

Parameters
s – scale factor for x-, y- and z-axis

rotate_x(*angle: float*)

Rotate mesh around x-axis about *angle* inplace.

Parameters
angle – rotation angle in radians

rotate_y(*angle: float*)

Rotate mesh around y-axis about *angle* inplace.

Parameters
angle – rotation angle in radians

rotate_z(*angle: float*)

Rotate mesh around z-axis about *angle* inplace.

Parameters
angle – rotation angle in radians

rotate_axis(*axis:* UVec, *angle: float*)

Rotate mesh around an arbitrary axis located in the origin (0, 0, 0) about *angle*.

Parameters

- **axis** – rotation axis as Vec3
- **angle** – rotation angle in radians

6.11.9 MeshVertexMerger

Same functionality as *MeshBuilder*, but created meshes with unique vertices and no doublets, but *MeshVertexMerger* needs extra memory for bookkeeping and also does not support transformations. The location of the merged vertices is the location of the first vertex with the same key.

This class is intended as intermediate object to create compact meshes and convert them to *MeshTransformer* objects to apply transformations:

```
mesh = MeshVertexMerger()

# create your mesh
mesh.add_face(...)

# convert mesh to MeshTransformer object
return MeshTransformer.from_builder(mesh)
```

class ezdxf.render.**MeshVertexMerger**(*precision: int = 6*)

Subclass of *MeshBuilder*

Mesh with unique vertices and no doublets, but needs extra memory for bookkeeping.

MeshVertexMerger creates a key for every vertex by rounding its components by the Python `round()` function and a given *precision* value. Each vertex with the same key gets the same vertex index, which is the index of first vertex with this key, so all vertices with the same key will be located at the location of this first vertex. If you want an average location of all vertices with the same key use the *MeshAverageVertexMerger* class.

Parameters
precision – floating point precision for vertex rounding

6.11.10 MeshAverageVertexMerger

This is an extended version of *MeshVertexMerger*. The location of the merged vertices is the average location of all vertices with the same key, this needs extra memory and runtime in comparison to *MeshVertexMerger* and this class also does not support transformations.

class ezdxf.render.**MeshAverageVertexMerger**(*precision: int = 6*)

Subclass of *MeshBuilder*

Mesh with unique vertices and no doublets, but needs extra memory for bookkeeping and runtime for calculation of average vertex location.

MeshAverageVertexMerger creates a key for every vertex by rounding its components by the Python `round()` function and a given *precision* value. Each vertex with the same key gets the same vertex index, which is the index of first vertex with this key, the difference to the *MeshVertexMerger* class is the calculation of the average location for all vertices with the same key, this needs extra memory to keep track of the count of vertices for each key and extra runtime for updating the vertex location each time a vertex with an existing key is added.

Parameters
precision – floating point precision for vertex rounding

class ezdxf.render.mesh.**EdgeStat**(*count: int*, *balance: int*)

Named tuple of edge statistics.

count

how often the edge *(a, b)* is used in faces as *(a, b)* or *(b, a)*

balance

count of edges *(a, b)* - count of edges *(b, a)* and should be 0 in "healthy" closed surfaces, if the balance is not 0, maybe doubled coincident faces exist or faces may have mixed clockwise and counter-clockwise vertex orders

6.11.11 MeshBuilder Helper Classes

class ezdxf.render.**MeshDiagnose**

Diagnose tool which can be used to analyze and detect errors of *MeshBuilder* objects like topology errors for closed surfaces. The object contains cached values, which do not get updated if the source mesh will be changed!

Note: There exist no tools in *ezdxf* to repair broken surfaces, but you can use the *ezdxf.addons.meshex* addon to exchange meshes with the open source tool MeshLab.

Create an instance of this tool by the *MeshBuilder.diagnose()* method.

property bbox: *BoundingBox*

Returns the *BoundingBox* of the mesh. (cached data)

property edge_stats: Dict[Tuple[int, int], *EdgeStat*]

Returns the edge statistics as a `dict`. The dict-key is the edge as tuple of two vertex indices *(a, b)* where *a* is always smaller than *b*. The dict-value is an `EdgeStat` tuple of edge count and edge balance, see `EdgeStat` for the definition of edge count and edge balance. (cached data)

property euler_characteristic: int

Returns the Euler characteristic: https://en.wikipedia.org/wiki/Euler_characteristic

This number is always 2 for convex polyhedra.

property face_normals: Sequence[*Vec3*]

Returns all face normal vectors as sequence. The `NULLVEC` instance is used as normal vector for degenerated faces. (cached data)

property faces: Sequence[Sequence[int]]

Sequence of faces as `Sequence[int]`

property is_closed_surface: bool

Returns `True` if the mesh has a closed surface. This method does not require a unified face orientation. If multiple separated meshes are present the state is only `True` if **all** meshes have a closed surface. (cached data)

Returns `False` for non-manifold meshes.

property is_edge_balance_broken: bool

Returns `True` if the edge balance is broken, this indicates a topology error for closed surfaces. A non-broken edge balance reflects that each edge connects two faces, where the edge is clockwise oriented in the first face and counter-clockwise oriented in the second face. A broken edge balance indicates possible topology errors like mixed face vertex orientations or a non-manifold mesh where an edge connects more than two faces. (cached data)

property is_manifold: bool

Returns `True` if all edges have an edge count < 3. (cached data)

A non-manifold mesh has edges with 3 or more connected faces.

property n_edges: int

Returns the unique edge count. (cached data)

property n_faces: int

Returns the face count.

property n_vertices: int

Returns the vertex count.

property vertices: Sequence[*Vec3*]

Sequence of mesh vertices as *Vec3* instances

centroid() → *Vec3*

Returns the centroid of all vertices. (center of mass)

estimate_face_normals_direction() → float

Returns the estimated face-normals direction as `float` value in the range [-1.0, 1.0] for a closed surface.

This heuristic works well for simple convex hulls but struggles with more complex structures like a torus (doughnut).

A counter-clockwise (ccw) vertex arrangement for outward pointing faces is assumed but a clockwise (cw) arrangement works too but the return values are reversed.

The closer the value to 1.0 (-1.0 for cw) the more likely all normals pointing outwards from the surface.

The closer the value to -1.0 (1.0 for cw) the more likely all normals pointing inwards from the surface.

There are no exact confidence values if all faces pointing outwards, here some examples for surfaces created by *ezdxf.render.forms* functions:

- *cube()* returns 1.0
- *cylinder()* returns 0.9992

- *sphere()* returns 0.9994
- *cone()* returns 0.9162
- *cylinder()* with all hull faces pointing outwards but caps pointing inwards returns 0.7785 but the property *is_edge_balance_broken* returns `True` which indicates the mixed vertex orientation
- and the estimation of 0.0469 for a *torus()* is barely usable

has_non_planar_faces() → bool

Returns `True` if any face is non-planar.

surface_area() → float

Returns the surface area.

total_edge_count() → int

Returns the total edge count of all faces, shared edges are counted separately for each face. In closed surfaces this count should be 2x the unique edge count *n_edges*. (cached data)

unique_edges() → Iterable[Tuple[int, int]]

Yields the unique edges of the mesh as int 2-tuples. (cached data)

volume() → float

Returns the volume of a closed surface or 0 otherwise.

> **Warning:** The face vertices have to be in counter-clockwise order, this requirement is not checked by this method.
>
> The result is not correct for multiple separated meshes in a single MeshBuilder object!!!

class ezdxf.render.**FaceOrientationDetector**(*mesh: MeshBuilder*, *reference: int = 0*)

Helper class for face orientation and face normal vector detection. Use the method *MeshBuilder.face_orientation_detector()* to create an instance.

The face orientation detector classifies the faces of a mesh by their forward or backward orientation. The forward orientation is defined by a reference face, which is the first face of the mesh by default and this orientation is not necessarily outwards.

This class has some overlapping features with *MeshDiagnose* but it has a longer setup time and needs more memory than *MeshDiagnose*.

Parameters

- **mesh** – source mesh as *MeshBuilder* object
- **reference** – index of the reference face

is_manifold

`True` if all edges have an edge count < 3. A non-manifold mesh has edges with 3 or more connected faces.

property all_reachable: bool

Returns `True` if all faces are reachable from the reference face same as property *is_single_mesh*.

property count: tuple[int, int]

Returns the count of forward and backward oriented faces.

property backward_faces: Iterator[Sequence[int]]

Yields all backward oriented faces.

property forward_faces: Iterator[Sequence[int]]

Yields all forward oriented faces.

property has_uniform_face_normals: bool

Returns `True` if all reachable faces are forward oriented according to the reference face.

property is_closed_surface: bool

Returns `True` if the mesh has a closed surface. This method does not require a unified face orientation. If multiple separated meshes are present the state is only `True` if **all** meshes have a closed surface.

Returns `False` for non-manifold meshes.

property is_single_mesh: bool

Returns `True` if only a single mesh is present same as property *all_reachable*.

classify_faces(*reference: int = 0*) → None

Detect the forward and backward oriented faces.

The forward and backward orientation has to be defined by a *reference* face.

is_reference_face_pointing_outwards() → bool

Returns `True` if the normal vector of the reference face is pointing outwards. This works only for meshes with unified faces which represent a closed surfaces, and it's a time-consuming calculation!

6.11.12 Trace

This module provides tools to create banded lines like LWPOLYLINE with width information. Path rendering as quadrilaterals: *Trace*, *Solid* or *Face3d*.

class ezdxf.render.trace.**TraceBuilder**

Sequence of 2D banded lines like polylines with start- and end width or curves with start- and end width.

Note: Accepts 3D input, but z-axis is ignored. The *TraceBuilder* is a 2D only object and uses only the *OCS* coordinates!

---

abs_tol

Absolute tolerance for floating point comparisons

append(*trace: AbstractTrace*) → None

Append a new trace.

close()

Close multi traces by merging first and last trace, if linear traces.

faces() → Iterable[Tuple[*Vec2*, *Vec2*, *Vec2*, *Vec2*]]

Yields all faces as 4-tuples of *Vec2* objects in *OCS*.

faces_wcs(*ocs:* OCS, *elevation: float*) → Iterable[Sequence[*Vec3*]]

Yields all faces as 4-tuples of *Vec3* objects in *WCS*.

virtual_entities(*dxftype='TRACE'*, *dxfattribs=None*, *doc:* Drawing | *None = None*) → Iterable[Quadrilateral]

Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in *dxfattribs*.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Note: The *TraceBuilder* is a 2D only object and uses only the *OCS* coordinates!

Parameters

- **dxftype** – DXF type as string, "SOLID", "TRACE" or "3DFACE"
- **dxfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities
- **doc** – associated document

classmethod from_polyline(*polyline:* DXFGraphic, *segments: int = 64*) → *TraceBuilder*

Create a complete trace from a LWPOLYLINE or a 2D POLYLINE entity, the trace consist of multiple sub-traces if *bulge* values are present. Uses only the *OCS* coordinates!

Parameters

- **polyline** – *LWPolyline* or 2D *Polyline*
- **segments** – count of segments for bulge approximation, given count is for a full circle, partial arcs have proportional less segments, but at least 3

__len__()

__getitem__()

class ezdxf.render.trace.**LinearTrace**

Linear 2D banded lines like polylines with start- and end width.

Accepts 3D input, but z-axis is ignored.

abs_tol

Absolute tolerance for floating point comparisons

is_started

True if at least one station exist.

add_station(*point:* UVec, *start_width: float*, *end_width: float | None = None*) → None

Add a trace station (like a vertex) at location *point*, *start_width* is the width of the next segment starting at this station, *end_width* is the end width of the next segment.

Adding the last location again, replaces the actual last location e.g. adding lines (a, b), (b, c), creates only 3 stations (a, b, c), this is very important to connect to/from splines.

Parameters

- **point** – 2D location (vertex), z-axis of 3D vertices is ignored.
- **start_width** – start width of next segment
- **end_width** – end width of next segment

faces() → Iterable[Tuple[*Vec2*, *Vec2*, *Vec2*, *Vec2*]]

Yields all faces as 4-tuples of *Vec2* objects.

First and last miter is 90 degrees if the path is not closed, otherwise the intersection of first and last segment is taken into account, a closed path has to have explicit the same last and first vertex.

virtual_entities(*dxftype='TRACE'*, *dxfattribs=None*, *doc:* Drawing | *None* = *None*) → Iterable[Quadrilateral]

Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in *dxfattribs*.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Parameters

- **dxftype** – DXF type as string, "SOLID", "TRACE" or "3DFACE"
- **dxfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities
- **doc** – associated document

class ezdxf.render.trace.**CurvedTrace**

2D banded curves like arcs or splines with start- and end width.

Represents always only one curved entity and all miter of curve segments are perpendicular to curve tangents.

Accepts 3D input, but z-axis is ignored.

faces() → Iterable[Tuple[*Vec2*, *Vec2*, *Vec2*, *Vec2*]]

Yields all faces as 4-tuples of *Vec2* objects.

virtual_entities(*dxftype='TRACE'*, *dxfattribs=None*, *doc:* Drawing | *None* = *None*) → Iterable[Quadrilateral]

Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in *dxfattribs*.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Parameters

- **dxftype** – DXF type as string, "SOLID", "TRACE" or "3DFACE"
- **dxfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities
- **doc** – associated document

classmethod from_arc(*arc:* ConstructionArc, *start_width: float*, *end_width: float*, *segments: int = 64*) → *CurvedTrace*

Create curved trace from an arc.

Parameters

- **arc** – *ConstructionArc* object
- **start_width** – start width
- **end_width** – end width
- **segments** – count of segments for full circle (360 degree) approximation, partial arcs have proportional less segments, but at least 3

Raises

ValueError – if arc.radius <= 0

classmethod from_spline(*spline:* BSpline, *start_width: float*, *end_width: float*, *segments: int*) → *CurvedTrace*

Create curved trace from a B-spline.

Parameters

- **spline** – *BSpline* object

- **start_width** – start width
- **end_width** – end width
- **segments** – count of segments for approximation

6.11.13 Point Rendering

Helper function to render *Point* entities as DXF primitives.

ezdxf.render.point.**virtual_entities**(*point: Point*, *pdsize: float = 1*, *pdmode: int = 0*) → list[ezdxf.entities.dxfgfx.DXFGraphic]

Yields point graphic as DXF primitives LINE and CIRCLE entities. The dimensionless point is rendered as zero-length line!

Check for this condition:

```
e.dxftype() == 'LINE' and e.dxf.start.isclose(e.dxf.end)
```

if the rendering engine can't handle zero-length lines.

Parameters

- **point** – DXF POINT entity
- **pdsize** – point size in drawing units
- **pdmode** – point styling mode, see *Point* class

See also:

Go to *ezdxf.entities.Point* class documentation for more information about POINT styling modes.

6.11.14 MultiLeaderBuilder

These are helper classes to build *MultiLeader* entities in an easy way. The MultiLeader entity supports two kinds of content, for each exist a specialized builder class:

- *MultiLeaderMTextBuilder* for *MText* content
- *MultiLeaderBlockBuilder* for *Block* content

The usual steps of the building process are:

1. create entity by a factory method
   - *add_multileader_mtext()*
   - *add_multileader_block()*
2. set the content
   - *MultiLeaderMTextBuilder.set_content()*
   - *MultiLeaderBlockBuilder.set_content()*
   - *MultiLeaderBlockBuilder.set_attribute()*
3. set properties
   - *MultiLeaderBuilder.set_arrow_properties()*
   - *MultiLeaderBuilder.set_connection_properties()*

- *MultiLeaderBuilder.set_connection_types()*
- *MultiLeaderBuilder.set_leader_properties()*
- *MultiLeaderBuilder.set_mleader_style()*
- *MultiLeaderBuilder.set_overall_scaling()*

4. add one or more leader lines
   - *MultiLeaderBuilder.add_leader_line()*
5. finalize building process
   - *MultiLeaderBuilder.build()*

The *Tutorial for MultiLeader* shows how to use these helper classes in more detail.

class ezdxf.render.**MultiLeaderBuilder**

Abstract base class to build *MultiLeader* entities.

property context: MLeaderContext

Returns the context entity *MLeaderContext*.

property multileader: MultiLeader

Returns the *MultiLeader* entity.

add_leader_line(*side:* ConnectionSide, *vertices: Iterable[*Vec2*]*) → None

Add leader as iterable of vertices in render UCS coordinates (*WCS* by default).

Note: Vertical (top, bottom) and horizontal attachment sides (left, right) can not be mixed in a single entity - this is a limitation of the MULTILEADER entity.

Parameters

- **side** – connection side where to attach the leader line
- **vertices** – leader vertices

build(*insert:* Vec2, *rotation: float = 0.0*, *ucs:* UCS | *None = None*) → None

Compute the required geometry data. The construction plane is the xy-plane of the given render *UCS*.

Parameters

- **insert** – insert location for the content in render UCS coordinates
- **rotation** – content rotation angle around the render UCS z-axis in degrees
- **ucs** – the render *UCS*, default is the *WCS*

set_arrow_properties(*name: str = ''*, *size: float = 0.0*)

Set leader arrow properties all leader lines have the same arrow type.

The MULTILEADER entity is able to support multiple arrows, but this seems to be unsupported by CAD applications and is therefore also not supported by the builder classes.

set_connection_properties(*landing_gap: float = 0.0*, *dogleg_length: float = 0.0*)

Set the properties how to connect the leader line to the content.

The landing gap is the space between the content and the start of the leader line. The "dogleg" is the first line segment of the leader in the "horizontal" direction of the content.

set_connection_types(*left=HorizontalConnection.by_style*, *right=HorizontalConnection.by_style*, *top=VerticalConnection.by_style*, *bottom=VerticalConnection.by_style*)

Set the connection type for each connection side.

set_leader_properties(*color: int | Tuple[int, int, int] = colors.BYBLOCK*, *linetype: str = 'BYBLOCK'*, *lineweight: int = const.LINEWEIGHT_BYBLOCK*, *leader_type=LeaderType.straight_lines*)

Set leader line properties.

Parameters

- **color** – line color as *AutoCAD Color Index (ACI)* or RGB tuple
- **linetype** – as name string, e.g. "BYLAYER"
- **lineweight** – as integer value, see: *Lineweights*
- **leader_type** – straight lines of spline type

set_mleader_style(*style: MLeaderStyle*)

Reset base properties by `MLeaderStyle` properties. This also resets the content!

set_overall_scaling(*scale: float*)

Set the overall scaling factor for the whole entity, except for the leader line vertices!

Parameters
scale – scaling factor > 0.0

MultiLeaderMTextBuilder

Specialization of `MultiLeaderBuilder` to build `MultiLeader` with MTEXT content.

class ezdxf.render.**MultiLeaderMTextBuilder**

set_content(*content: str*, *color: int | Tuple[int, int, int] | None = None*, *char_height: float = 0.0*, *alignment:* TextAlignment *= TextAlignment.left*, *style: str = ''*)

Set MTEXT content.

Parameters

- **content** – MTEXT content as string
- **color** – block color as *AutoCAD Color Index (ACI)* or RGB tuple
- **char_height** – initial char height in drawing units
- **alignment** – `TextAlignment` - left, center, right
- **style** – name of `Textstyle` as string

quick_leader(*content: str*, *target:* Vec2, *segment1:* Vec2, *segment2:* Vec2 | *None = None*, *connection_type:* HorizontalConnection | VerticalConnection *= HorizontalConnection.middle_of_top_line*, *ucs:* UCS | *None = None*) → None

Creates a quick MTEXT leader. The *target* point defines where the leader points to. The *segment1* is the first segment of the leader line relative to the *target* point, *segment2* is an optional second line segment relative to the first line segment. The *connection_type* defines the type of connection (horizontal or vertical) and the MTEXT alignment (left, center or right). Horizontal connections are always left or right aligned, vertical connections are always center aligned.

Parameters

- **content** – MTEXT content string
- **target** – leader target point as `Vec2`
- **segment1** – first leader line segment as relative distance to *insert*
- **segment2** – optional second leader line segment as relative distance to first line segment
- **connection_type** – one of *HorizontalConnection* or *VerticalConnection*
- **ucs** – the rendering *UCS*, default is the *WCS*

MultiLeaderBlockBuilder

Specialization of *MultiLeaderBuilder* to build *MultiLeader* with BLOCK content.

class ezdxf.render.**MultiLeaderBlockBuilder**

property block_layout: *BlockLayout*

Returns the block layout.

property extents: *BoundingBox*

Returns the bounding box of the block.

set_content(*name: str*, *color: int | Tuple[int, int, int] = colors.BYBLOCK*, *scale: float = 1.0*, *alignment=BlockAlignment.center_extents*)

Set BLOCK content.

Parameters

- **name** – the block name as string
- **color** – block color as *AutoCAD Color Index (ACI)* or RGB tuple
- **scale** – the block scaling, not to be confused with overall scaling
- **alignment** – the block insertion point or the center of extents

set_attribute(*tag: str*, *text: str*, *width: float = 1.0*)

Add BLOCK attributes based on an ATTDEF entity in the block definition. All properties of the new created ATTRIB entity are defined by the template ATTDEF entity including the location.

Parameters

- **tag** – attribute tag name
- **text** – attribute content string
- **width** – width factor

Enums

class ezdxf.render.**LeaderType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The leader type.

none

straight_lines

splines

class ezdxf.render.**ConnectionSide**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The leader connection side.

Vertical (top, bottom) and horizontal attachment sides (left, right) can not be mixed in a single entity - this is a limitation of the MULTILEADER entity.

left

right

top

bottom

class ezdxf.render.**HorizontalConnection**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The horizontal leader connection type.

by_style

top_of_top_line

middle_of_top_line

middle_of_text

middle_of_bottom_line

bottom_of_bottom_line

bottom_of_bottom_line_underline

bottom_of_top_line_underline

bottom_of_top_line

bottom_of_top_line_underline_all

class ezdxf.render.**VerticalConnection**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The vertical leader connection type.

by_style

center

center_overline

class ezdxf.render.**TextAlignment**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The `MText` alignment type.

left

center

right

class ezdxf.render.**BlockAlignment**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The `Block` alignment type.

center_extents

insertion_point

6.11.15 Arrows

This module provides support for the AutoCAD standard arrow heads used in DIMENSION, LEADER and MULTILEADER entities. Library user don't have to use the *ARROWS* objects directly, but should know the arrow names stored in it as attributes. The arrow names should be accessed that way:

```
import ezdxf

arrow = ezdxf.ARROWS.closed_filled
```

ezdxf.render.arrows.**ARROWS**

Single instance of *_Arrows* to work with.

class ezdxf.render.arrows.**_Arrows**

Management object for standard arrows.

__acad__

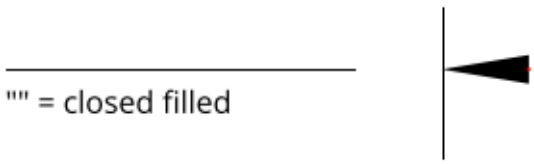
Set of AutoCAD standard arrow names.

__ezdxf__

Set of arrow names special to *ezdxf*.

architectural_tick

closed_filled



dot

dot_small

dot_blank

origin_indicator

origin_indicator_2

open

right_angle

open_30

closed

dot_smallblank

none

oblique

box_filled

box

closed_blank

datum_triangle_filled

datum_triangle

integral

ez_arrow

ez_arrow_blank

ez_arrow_filled

is_acad_arrow(*item: str*) → bool

Returns `True` if *item* is a standard AutoCAD arrow.

is_ezdxf_arrow(*item: str*) → bool

Returns `True` if *item* is a special *ezdxf* arrow.

insert_arrow(*layout: GenericLayoutType*, *name: str*, *insert:* UVec = *NULLVEC*, *size: float = 1.0*, *rotation: float = 0*, ***, *dxfattribs=None*) → *Vec2*

Insert arrow as block reference into *layout*.

render_arrow(*layout: GenericLayoutType*, *name: str*, *insert:* UVec = *NULLVEC*, *size: float = 1.0*, *rotation: float = 0*, ***, *dxfattribs=None*) → *Vec2*

Render arrow as basic DXF entities into *layout*.

virtual_entities(*name: str*, *insert:* UVec = *NULLVEC*, *size: float = 0.625*, *rotation: float = 0*, ***, *dxfattribs=None*) → Iterator[*DXFGraphic*]

Returns all arrow components as virtual DXF entities.

6.11.16 Hatching

This module provides rendering support for hatch patterns as used in *Hatch* and *MPolygon* entities.

High Level Functions

ezdxf.render.hatching.**hatch_entity**(*polygon: DXFPolygon*, *filter_text_boxes=True*) → Iterator[tuple[*Vec3*, *Vec3*]]

Yields the hatch pattern of the given HATCH or MPOLYGON entity as 3D lines. Each line is a pair of *Vec3* instances as start- and end vertex, points are represented as lines of zero length, which means the start vertex is equal to the end vertex.

The function yields nothing if *polygon* has a solid- or gradient filling or does not have a usable pattern assigned.

Parameters

- **polygon** – *Hatch* or *MPolygon* entity
- **filter_text_boxes** – ignore text boxes if `True`

ezdxf.render.hatching.**hatch_polygons**(*baseline:* HatchBaseLine, *polygons: Sequence[Sequence[*Vec2*]]*, *terminate: Callable[[], bool]* | *None = None*) → Iterator[*Line*]

Yields all pattern lines for all hatch lines generated by the given *HatchBaseLine*, intersecting the given 2D polygons as *Line* instances. The *polygons* should represent a single entity with or without holes, the order of the polygons and their winding orientation (cw or ccw) is not important. Entities which do not intersect or overlap should be handled separately!

Each polygon is a sequence of *Vec2* instances, they are treated as closed polygons even if the last vertex is not equal to the first vertex.

The hole detection is done by a simple inside/outside counting algorithm and far from perfect, but is able to handle ordinary polygons well.

The terminate function WILL BE CALLED PERIODICALLY AND should return `True` to terminate execution. This can be used to implement a timeout, which can be required if using a very small hatching distance, especially if you get the data from untrusted sources.

Parameters

- **baseline** – *HatchBaseLine*
- **polygons** – multiple sequences of `Vec2` instances of a single entity, the order of exterior- and hole paths and the winding orientation (cw or ccw) of paths is not important
- **terminate** – callback function which is called periodically and should return `True` to terminate the hatching function

ezdxf.render.hatching.**hatch_paths**(*baseline:* HatchBaseLine, *paths: Sequence[Path]*, *terminate: Callable[[], bool]* | *None = None*) → Iterator[*Line*]

Yields all pattern lines for all hatch lines generated by the given *HatchBaseLine*, intersecting the given 2D *Path* instances as *Line* instances. The paths are handled as projected into the xy-plane the z-axis of path vertices will be ignored if present.

Same as the *hatch_polygons()* function, but for *Path* instances instead of polygons build of vertices. This function **does not flatten** the paths into vertices, instead the real intersections of the Bézier curves and the hatch lines are calculated.

For more information see the docs of the *hatch_polygons()* function.

Parameters

- **baseline** – *HatchBaseLine*
- **paths** – sequence of *Path* instances of a single entity, the order of exterior- and hole paths and the winding orientation (cw or ccw) of the paths is not important
- **terminate** – callback function which is called periodically and should return `True` to terminate the hatching function

Classes

class ezdxf.render.hatching.**HatchBaseLine**(*origin:* Vec2, *direction:* Vec2, *offset:* Vec2, *line_pattern: list[float] | None = None*)

A hatch baseline defines the source line for hatching a geometry. A complete hatch pattern of a DXF entity can consist of one or more hatch baselines.

Parameters

- **origin** – the origin of the hatch line as *Vec2* instance
- **direction** – the hatch line direction as *Vec2* instance, must not (0, 0)
- **offset** – the offset of the hatch line origin to the next or to the previous hatch line
- **line_pattern** – line pattern as sequence of floats, see also *PatternRenderer*

Raises

- ***HatchLineDirectionError*** – hatch baseline has no direction, (0, 0) vector
- ***DenseHatchingLinesError*** – hatching lines are too narrow

hatch_line(*distance: float*) → *HatchLine*

Returns the *HatchLine* at the given signed *distance*.

pattern_renderer(*distance: float*) → *PatternRenderer*

Returns the *PatternRenderer* for the given signed *distance*.

signed_distance(*point:* Vec2) → float

Returns the signed normal distance of the given *point* from this hatch baseline.

class ezdxf.render.hatching.**HatchLine**(*origin:* Vec2, *direction:* Vec2, *distance: float*)

Represents a single hatch line.

Parameters

- **origin** – the origin of the hatch line as *Vec2* instance
- **direction** – the hatch line direction as *Vec2* instance, must not (0, 0)
- **distance** – the normal distance to the base hatch line as float

intersect_line(*a:* Vec2, *b:* Vec2, *dist_a: float*, *dist_b: float*) → *Intersection*

Returns the *Intersection* of this hatch line and the line defined by the points *a* and *b*. The arguments *dist_a* and *dist_b* are the signed normal distances of the points *a* and *b* from the hatch baseline. The normal distances from the baseline are easy to calculate by the *HatchBaseLine.signed_distance()* method and allow a fast intersection calculation by a simple point interpolation.

Parameters

- **a** – start point of the line as *Vec2* instance
- **b** – end point of the line as *Vec2* instance
- **dist_a** – normal distance of point *a* to the hatch baseline as float
- **dist_b** – normal distance of point *b* to the hatch baseline as float

intersect_cubic_bezier_curve(*curve:* Bezier4P) → Sequence[*Intersection*]

Returns 0 to 3 *Intersection* points of this hatch line with a cubic Bèzier curve.

Parameters

curve – the cubic Bèzier curve as *ezdxf.math.Bezier4P* instance

class ezdxf.render.hatching.**PatternRenderer**(*hatch_line:* HatchLine, *pattern: Sequence[float]*)

The hatch pattern of a DXF entity has one or more *HatchBaseLine* instances with an origin, direction, offset and line pattern. The *PatternRenderer* for a certain distance from the baseline has to be acquired from the *HatchBaseLine* by the *pattern_renderer()* method.

The origin of the hatch line is the starting point of the line pattern. The offset defines the origin of the adjacent hatch line and doesn't have to be orthogonal to the hatch line direction.

Line Pattern

The line pattern is a sequence of floats, where a value > 0.0 is a dash, a value < 0.0 is a gap and value of 0.0 is a point.

Parameters

- **hatch_line** – *HatchLine*
- **pattern** – the line pattern as sequence of float values

render(*start:* Vec2, *end:* Vec2) → Iterator[tuple[*ezdxf.math._vector.Vec2*, *ezdxf.math._vector.Vec2*]]

Yields the pattern lines as pairs of *Vec2* instances from the start- to the end point on the hatch line. For points the start- and end point are the same *Vec2* instance and can be tested by the `is` operator.

The start- and end points should be located collinear at the hatch line of this instance, otherwise the points a projected onto this hatch line.

class ezdxf.render.hatching.**Intersection**(*type:* IntersectionType = *IntersectionType.NONE*, *p0:* Vec2 = *Vec2(nan, nan)*, *p1:* Vec2 = *Vec2(nan, nan)*)

Represents an intersection.

type

intersection type as *IntersectionType* instance

p0

(first) intersection point as *Vec2* instance

p1

second intersection point as *Vec2* instance, only if *type* is COLLINEAR

class ezdxf.render.hatching.**IntersectionType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

NONE

no intersection

REGULAR

regular intersection point at a polygon edge or a Bèzier curve

START

intersection point at the start vertex of a polygon edge

END

intersection point at the end vertex of a polygon edge

COLLINEAR

intersection is collinear to a polygon edge

class ezdxf.render.hatching.**Line**(*start: 'Vec2'*, *end: 'Vec2'*, *distance: 'float'*)

start
start point as *Vec2* instance

end
end point as *Vec2* instance

distance
signed normal distance to the *HatchBaseLine*

Helper Functions

ezdxf.render.hatching.**hatch_boundary_paths**(*polygon: DXFPolygon*, *filter_text_boxes=True*) → list[*Path*]

Returns the hatch boundary paths as *ezdxf.path.Path* instances of HATCH and MPOLYGON entities. Ignores text boxes if argument *filter_text_boxes* is `True`.

ezdxf.render.hatching.**hatch_line_distances**(*point_distances: Sequence[float]*, *normal_distance: float*) → list[float]

Returns all hatch line distances in the range of the given point distances.

ezdxf.render.hatching.**pattern_baselines**(*polygon: DXFPolygon*) → Iterator[*HatchBaseLine*]

Yields the hatch pattern baselines of HATCH and MPOLYGON entities as *HatchBaseLine* instances.

Exceptions

class ezdxf.render.hatching.**HatchingError**

Base exception class of the `hatching` module.

class ezdxf.render.hatching.**HatchLineDirectionError**

Hatching direction is undefined or a (0, 0) vector.

class ezdxf.render.hatching.**DenseHatchingLinesError**

Very small hatching distance which creates too many hatching lines.

6.12 Add-ons

6.12.1 Drawing / Export Addon

This add-on provides the functionality to render a DXF document to produce a rasterized or vector-graphic image which can be saved to a file or viewed interactively depending on the backend being used.

The module provides two example scripts in the folder `examples/addons/drawing` which can be run to save rendered images to files or view an interactive visualisation.

```
$ ./draw_cad.py --supported_formats
# will list the file formats supported by the matplotlib backend.
# Many formats are supported including vector graphics formats
# such as pdf and svg

$ ./draw_cad.py <my_file.dxf> --out image.png

# draw a layout other than the model space
```

(continues on next page)

(continued from previous page)

```
$ ./draw_cad.py <my_file.dxf> --layout Layout1 --out image.png

# opens a GUI application to view CAD files
$ ./cad_viewer.py
```

See also:

How-to section for the FAQ about the *Drawing Add-on*.

Design

The implementation of the `drawing` add-on is divided into a frontend and multiple backends. The frontend handles the translation of DXF features and properties into simplified structures, which are then processed by the backends.

Common Limitations to all Backends

- rich text formatting of the MTEXT entity is close to AutoCAD but not pixel perfect
- relative size of POINT entities cannot be replicated exactly
- rendering of ACIS entities is not supported
- no 3D rendering engine, therefore:
  - 3D entities are projected into the xy-plane and 3D text is not supported
  - only top view rendering of the modelspace
  - VIEWPORTS are always rendered as top view
  - no visual style support
- only basic support for:
  - infinite lines (rendered as lines with a finite length)
  - OLE2FRAME entities (rendered as rectangles)
  - vertical text (will render as horizontal text)
  - rendering of additional MTEXT columns may be incorrect

MatplotlibBackend

class ezdxf.addons.drawing.matplotlib.**MatplotlibBackend**(*ax*, ***, *adjust_figure=True*, *font=FontProperties()*, *use_text_cache=True*)

Backend which uses the `Matplotlib` package for image export.

Parameters

- **ax** – drawing canvas as `matplotlib.pyplot.Axes` object
- **adjust_figure** – automatically adjust the size of the parent `matplotlib.pyplot.Figure` to display all content
- **font** – default font properties
- **use_text_cache** – use caching for text path rendering

The `MatplotlibBackend` is used by the *Draw* command of the *ezdxf* launcher.

Limitations

- the text path rendering is different to AutoCAD, therefore text placement and wrapping may appear slightly different
- the SVG export of dimensionless POINT entities is rendered as circles
- has no VIEWPORT clipping support (yet?) and is therefore not very suitable for exporting paperspace layouts

Example for the usage of the `Matplotlib` backend:

```
import sys
import matplotlib.pyplot as plt
from ezdxf import recover
from ezdxf.addons.drawing import RenderContext, Frontend
from ezdxf.addons.drawing.matplotlib import MatplotlibBackend

# Safe loading procedure (requires ezdxf v0.14):
try:
    doc, auditor = recover.readfile('your.dxf')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)

# The auditor.errors attribute stores severe errors,
# which may raise exceptions when rendering.
if not auditor.has_errors:
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1])
    ctx = RenderContext(doc)
    out = MatplotlibBackend(ax)
    Frontend(ctx, out).draw_layout(doc.modelspace(), finalize=True)
    fig.savefig('your.png', dpi=300)
```

Simplified render workflow but with less control:

```
from ezdxf import recover
from ezdxf.addons.drawing import matplotlib

# Exception handling left out for compactness:
doc, auditor = recover.readfile('your.dxf')
if not auditor.has_errors:
    matplotlib.qsave(doc.modelspace(), 'your.png')
```

ezdxf.addons.drawing.matplotlib.**qsave**(*layout: Layout*, *filename: str | PathLike*, ***, *bg: str | None = None*, *fg: str | None = None*, *dpi: int = 300*, *backend: str = 'agg'*, *config:* Configuration *| None = None*, *filter_func: Callable[[DXFGraphic], bool] | None = None*, *size_inches: tuple[float, float] | None = None*) → None

Quick and simplified render export by matplotlib.

Parameters

- **layout** – modelspace or paperspace layout to export

- **filename** – export filename, file extension determines the format e.g. "image.png" to save in PNG format.
- **bg** – override default background color in hex format #RRGGBB or #RRGGBBAA, e.g. use bg="#FFFFFF00" to get a transparent background and a black foreground color (ACI=7), because a white background #FFFFFF gets a black foreground color or vice versa bg="#00000000" for a transparent (black) background and a white foreground color.
- **fg** – override default foreground color in hex format #RRGGBB or #RRGGBBAA, requires also *bg* argument. There is no explicit foreground color in DXF defined (also not a background color), but the ACI color 7 has already a variable color value, black on a light background and white on a dark background, this argument overrides this (ACI=7) default color value.
- **dpi** – image resolution (dots per inches).
- **size_inches** – paper size in inch as *(width, height)* tuple, which also defines the size in pixels = (*width* * *dpi*) x (*height* * *dpi*). If *width* or *height* is 0.0 the value is calculated by the aspect ratio of the drawing.
- **backend** – the matplotlib rendering backend to use (agg, cairo, svg etc) (see documentation for matplotlib.use() for a complete list of backends)
- **config** – drawing parameters
- **filter_func** – filter function which takes a DXFGraphic object as input and returns `True` if the entity should be drawn or `False` if the entity should be ignored

PyQtBackend

class ezdxf.addons.drawing.pyqt.**PyQtBackend**(*scene=None*, ***, *extra_lineweight_scaling=2.0*, *use_text_cache=True*)

Backend which uses the `PySide6` package to implement an interactive viewer. The `PyQt5` package can be used as fallback if the `PySide6` package is not available.

Parameters

- **scene** – drawing canvas of type `QtWidgets.QGraphicsScene`, if `None` a new canvas will be created
- **extra_lineweight_scaling** – compared to other backends, PyQt draws lines which appear thinner
- **use_text_cache** – use caching for text path rendering

The `PyQtBackend` is used by the *View* command of the *ezdxf* launcher.

Limitations

- the text path rendering is different to AutoCAD, therefore text placement and wrapping may appear slightly different

See also:

The qtviewer.py module implements the core of a simple DXF viewer and the cad_viewer.py example is a skeleton to show how to launch the `CADViewer` class.

PillowBackend

class ezdxf.addons.drawing.pillow.**PillowBackend**(*region: AbstractBoundingBox*, *image_size: tuple[int, int] | None = None*, *resolution: float = 1.0*, *margin: int = 10*, *dpi: int = 300*, *oversampling: int = 1*, *text_mode=TextMode.OUTLINE*)

Backend which uses the `Pillow` package for image export.

For linetype support configure the line_policy in the frontend as ACCURATE.

Parameters

- **region** – output region of the layout in DXF drawing units
- **image_size** – image output size in pixels or `None` to be calculated by the region size and the *resolution*
- **margin** – image margin in pixels, same margin for all four borders
- **resolution** – pixels per DXF drawing unit, e.g. a resolution of 100 for the drawing unit "meter" means, each pixel represents an area of 1cm x 1cm (1m is 100cm). If the *image_size* is given the *resolution* is calculated automatically
- **dpi** – output image resolution in dots per inch. The pixel width of lines is determined by the DXF lineweight (in mm) and this image resolution (dots/pixels per inch). The line width is independent of the drawing scale!
- **oversampling** – multiplier of the final image size to define the render canvas size (e.g. 1, 2, 3, …), the final image will be scaled down by the LANCZOS method
- **text_mode** – text rendering mode
  - IGNORE: do not draw text
  - PLACEHOLDER: draw text as filled rectangles
  - OUTLINE: draw text as outlines (recommended)
  - FILLED: simulate text filling by hatching the text outline with dense lines - has some issues

The `PillowBackend` is used by the *Pillow* command of the *ezdxf* launcher. The pillow.py example script is the development prototype for the *Pillow* command.

Limitations

- text paths are generated by `matplotlib`
- the text path rendering is different to AutoCAD, therefore text placement and wrapping may appear slightly different
- no real solid fill for HATCH entities and text glyphs; simulated by a dense line pattern filling, to improve this, a triangulation algorithm with support for nested holes (hole in hole in hole …) is required

Configuration

Additional options for the drawing add-on can be passed by the *config* argument of the `Frontend` constructor `__init__()`. Not every option will be supported by all backends.

class ezdxf.addons.drawing.config.**Configuration**(*pdsize: int | None*, *pdmode: int | None*, *measurement:* Measurement *| None*, *show_defpoints: bool*, *proxy_graphic_policy:* ProxyGraphicPolicy, *line_policy:* LinePolicy, *hatch_policy:* HatchPolicy, *infinite_line_length: float*, *lineweight_scaling: float*, *min_lineweight: float | None*, *min_dash_length: float*, *max_flattening_distance: float*, *circle_approximation_count: int*, *hatching_timeout: float*)

Configuration options for the `drawing` add-on.

pdsize

the size to draw POINT entities (in drawing units) set to None to use the $PDSIZE value from the dxf document header

0	5% of draw area height
<0	Specifies a percentage of the viewport size
>0	Specifies an absolute size
None	use the $PDMODE value from the dxf document header

Type
int | None

pdmode

point styling mode (see POINT documentation)

see *Point* class documentation

Type
int | None

measurement

whether to use metric or imperial units as enum *ezdxf.enums.Measurement*

0	use imperial units (in, ft, yd, …)
1	use metric units (ISO meters)
None	use the $MEASUREMENT value from the dxf document header

Type
ezdxf.enums.Measurement | None

show_defpoints

whether to show or filter out POINT entities on the defpoints layer

Type
bool

proxy_graphic_policy

the action to take when a proxy graphic is encountered

Type
ezdxf.addons.drawing.config.ProxyGraphicPolicy

line_policy

the method to use when drawing styled lines (eg dashed, dotted etc)

Type
ezdxf.addons.drawing.config.LinePolicy

hatch_policy

the method to use when drawing HATCH entities

Type
ezdxf.addons.drawing.config.HatchPolicy

infinite_line_length

the length to use when drawing infinite lines

Type
float

lineweight_scaling

multiplies every lineweight by this factor; set this factor to 0.0 for a constant minimum line width defined by the `min_lineweight` setting for all lineweights; the correct DXF lineweight often looks too thick in SVG, so setting a factor < 1 can improve the visual appearance

Type
float

min_lineweight

the minimum line width in 1/300 inch; set to `None` for let the backend choose.

Type
float | None

min_dash_length

the minimum length for a dash when drawing a styled line (default value is arbitrary)

Type
float

max_flattening_distance

Max flattening distance in drawing units see Path.flattening documentation. The backend implementation should calculate an appropriate value, like 1 screen- or paper pixel on the output medium, but converted into drawing units. Sets Path() approximation accuracy

Type
float

circle_approximation_count

Approximate a full circle by *n* segments, arcs have proportional less segments. Only used for approximation of arcs in banded polylines.

Type
int

hatching_timeout

hatching timeout for a single entity, very dense hatching patterns can cause a very long execution time, the default timeout for a single entity is 30 seconds.

Type
float

defaults()

Returns a frozen *Configuration* object with default values.

with_changes()

Returns a new frozen *Configuration* object with modified values.

Usage:

```
my_config = Configuration.defaults()
my_config = my_config.with_changes(lineweight_scaling=2)
```

LinePolicy

class ezdxf.addons.drawing.config.**LinePolicy**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

SOLID

draw all lines as solid regardless of the linetype style

APPROXIMATE

use the closest approximation available to the backend for rendering styled lines

ACCURATE

analyse and render styled lines as accurately as possible. This approach is slower and is not well suited to interactive applications.

HatchPolicy

class ezdxf.addons.drawing.config.**HatchPolicy**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The action to take when a HATCH entity is encountered

IGNORE

do not show HATCH entities at all

SHOW_OUTLINE

show only the outline of HATCH entities

SHOW_SOLID

show HATCH entities but draw with solid fill regardless of the pattern

ProxyGraphicPolicy

class ezdxf.addons.drawing.config.**ProxyGraphicPolicy**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The action to take when an entity with a proxy graphic is encountered

---

Note: To get proxy graphics support proxy graphics have to be loaded: Set the global option *ezdxf.options.load_proxy_graphics* to `True`, which is the default value.

This can not prevent drawing proxy graphic inside of blocks, because this is outside of the domain of the drawing add-on!

IGNORE

do not display proxy graphics (skip_entity will be called instead)

SHOW

if the entity cannot be rendered directly (eg if not implemented) but a proxy is present: display the proxy

PREFER

display proxy graphics even for entities where direct rendering is available

Properties

class ezdxf.addons.drawing.properties.**Properties**

An implementation agnostic representation of DXF entity properties like color and linetype. These properties represent the actual values after resolving all DXF specific rules like "by layer", "by block" and so on.

color

The actual color value of the DXF entity as "#RRGGBB" or "#RRGGBBAA" string. An alpha value of "00" is opaque and "ff" is fully transparent.

rgb

RGB values extract from the *color* value as tuple of integers.

luminance

Perceived luminance calculated from the *color* value as float in the range [0.0, 1.0].

linetype_name

The actual linetype name as string like "CONTINUOUS"

linetype_pattern

The simplified DXF linetype pattern as tuple of floats, all line elements and gaps are values greater than 0.0 and 0.0 represents a point. Line or point elements do always alternate with gap elements: line-gap-line-gap-point-gap and the pattern always ends with a gap. The continuous line is an empty tuple.

linetype_scale

The scaling factor as float to apply to the *linetype_pattern*.

lineweight

The absolute lineweight to render in mm as float.

is_visible

Visibility flag as bool.

layer
The actual layer name the entity resides on as UPPERCASE string.

font
The `FontFace` used for text rendering or `None`.

filling
The actual `Filling` properties of the entity or `None`.

units
The actual drawing units as *`InsertUnits`* enum.

LayerProperties

class ezdxf.addons.drawing.properties.**LayerProperties**
Actual layer properties, inherits from class `Properties`.

is_visible
Modified meaning: whether entities belonging to this layer should be drawn

layer
Modified meaning: stores real layer name (mixed case)

LayoutProperties

class ezdxf.addons.drawing.properties.**LayoutProperties**
Actual layout properties.

name
Layout name as string

units
Layout units as *`InsertUnits`* enum.

property LayoutProperties.**background_color: str**
Returns the default layout background color.

property LayoutProperties.**default_color: str**
Returns the default layout foreground color.

property LayoutProperties.**has_dark_background: bool**
Returns `True` if the actual background-color is "dark".

LayoutProperties.**set_colors**(*bg: str*, *fg: str | None = None*) → None
Setup default layout colors.

Required color format "#RRGGBB" or including alpha transparency "#RRGGBBAA".

RenderContext

class ezdxf.addons.drawing.properties.**RenderContext**(*doc: Drawing | None = None*, ***, *ctb: str = ''*, *export_mode: bool = False*)

The render context for the given DXF document. The *RenderContext* resolves the properties of DXF entities from the context they reside in to actual values like RGB colors, transparency, linewidth and so on.

A given *ctb* file (plot style file) overrides the default properties for all layouts, which means the plot style table stored in the layout is always ignored.

Parameters

- **doc** – DXF document
- **ctb** – path to a plot style table
- **export_mode** – Whether to render the document as it would look when exported (plotted) by a CAD application to a file such as pdf, or whether to render the document as it would appear inside a CAD application.

resolve_aci_color(*aci: int*, *resolved_layer: str*) → str

Resolve the *aci* color as hex color string: "#RRGGBB"

resolve_all(*entity: DXFGraphic*) → *Properties*

Resolve all properties of *entity*.

resolve_color(*entity: DXFGraphic*, ***, *resolved_layer: str | None = None*) → str

Resolve the rgb-color of *entity* as hex color string: "#RRGGBB" or "#RRGGBBAA".

resolve_filling(*entity: DXFGraphic*) → Filling | None

Resolve filling properties (SOLID, GRADIENT, PATTERN) of *entity*.

resolve_font(*entity: DXFGraphic*) → *FontFace* | None

Resolve the text style of *entity* to a font name. Returns `None` for the default font.

resolve_layer(*entity: DXFGraphic*) → str

Resolve the layer of *entity*, this is only relevant for entities inside of block references.

resolve_layer_properties(*layer: Layer*) → *LayerProperties*

Resolve layer properties.

resolve_linetype(*entity: DXFGraphic*, ***, *resolved_layer: str | None = None*) → tuple[str, Sequence[float]]

Resolve the linetype of *entity*. Returns a tuple of the linetype name as upper-case string and the simplified linetype pattern as tuple of floats.

resolve_lineweight(*entity: DXFGraphic*, ***, *resolved_layer: str | None = None*) → float

Resolve the lineweight of *entity* in mm.

DXF stores the lineweight in mm times 100 (e.g. 0.13mm = 13). The smallest line weight is 0 and the biggest line weight is 211. The DXF/DWG format is limited to a fixed value table, see: `ezdxf.lldxf.const.VALID_DXF_LINEWEIGHTS`

CAD applications draw lineweight 0mm as an undefined small value, to prevent backends to draw nothing for lineweight 0mm the smallest return value is 0.01mm.

resolve_units() → *InsertUnits*

resolve_visible(*entity: DXFGraphic*, *, *resolved_layer: str | None = None*) → bool

Resolve the visibility state of *entity*. Returns `True` if *entity* is visible.

set_current_layout(*layout:* Layout, *ctb: str = ''*)

Set the current layout and update layout specific properties.

set_layer_properties_override(*func: Callable[[Sequence[*LayerProperties*]], None] | None = None*)

The function *func* is called with the current layer properties as argument after resetting them, so the function can override the layer properties.

The `RenderContext` class can be used isolated from the `drawing` add-on to resolve DXF properties.

Frontend

class ezdxf.addons.drawing.frontend.**Frontend**(*ctx:* RenderContext, *out:* BackendInterface, *config:* Configuration = *Configuration.defaults()*, *bbox_cache:* ezdxf.bbox.Cache = *None*)

Drawing frontend, responsible for decomposing entities into graphic primitives and resolving entity properties.

By passing the bounding box cache of the modelspace entities can speed up paperspace rendering, because the frontend can filter entities which are not visible in the VIEWPORT. Even passing in an empty cache can speed up rendering time when multiple viewports need to be processed.

Parameters

- **ctx** – the properties relevant to rendering derived from a DXF document
- **out** – the backend to draw to
- **config** – settings to configure the drawing frontend and backend
- **bbox_cache** – bounding box cache of the modelspace entities or an empty cache which will be filled dynamically when rendering multiple viewports or `None` to disable bounding box caching at all

log_message(*message: str*)

Log given message - override to alter behavior.

skip_entity(*entity: DXFEntity*, *msg: str*) → None

Called for skipped entities - override to alter behavior.

override_properties(*entity: DXFGraphic*, *properties:* Properties) → None

The *override_properties()* filter can change the properties of an entity independent of the DXF attributes.

This filter has access to the DXF attributes by the *entity* object, the current render context, and the resolved properties by the *properties* object. It is recommended to modify only the *properties* object in this filter.

draw_layout(*layout: Layout*, *finalize: bool = True*, *, *filter_func: Callable[[DXFGraphic], bool] | None = None*, *layout_properties:* LayoutProperties | *None = None*) → None

Draw all entities of the given *layout*.

Draws the entities of the layout in the default or redefined redraw-order and calls the `finalize()` method of the backend if requested. The default redraw order is the ascending handle order not the order the entities are stored in the layout.

The method skips invisible entities and entities for which the given filter function returns `False`.

Parameters

- **layout** – layout to draw of type *Layout*
- **finalize** – `True` if the `finalize()` method of the backend should be called automatically
- **filter_func** – function to filter DXf entities, the function should return `False` if a given entity should be ignored
- **layout_properties** – override the default layout properties

BackendInterface

class ezdxf.addons.drawing.backend.**BackendInterface**

The public interface definition for the rendering backend.

For more information read the source code: backend.py

Backend

class ezdxf.addons.drawing.backend.**Backend**

Abstract base class for concrete backend implementations and implements some default features.

For more information read the source code: backend.py

Details

The rendering is performed in two stages. The frontend traverses the DXF document structure, converting each encountered entity into primitive drawing commands. These commands are fed to a backend which implements the interface: *Backend*.

Although the resulting images will not be pixel-perfect with AutoCAD (which was taken as the ground truth when developing this add-on) great care has been taken to achieve similar behavior in some areas:

- The algorithm for determining color should match AutoCAD. However, the color palette is not stored in the DXF file, so the chosen colors may be different to what is expected. The *RenderContext* class supports passing a plot style table (*CTB*-file) as custom color palette but uses the same palette as AutoCAD by default.
- Text rendering is quite accurate, text positioning, alignment and word wrapping are very faithful. Differences may occur if a different font from what was used by the CAD application but even in that case, for supported backends, measurements are taken of the font being used to match text as closely as possible.
- Visibility determination (based on which layers are visible) should match AutoCAD

See also:

- draw_cad.py for a simple use of this module
- cad_viewer.py for an advanced use of this module
- *Notes on Rendering DXF Content* for additional behaviours documented during the development of this add-on.

6.12.2 Geo Interface

Intended Usage

The intended usage of the *ezdxf.addons.geo* module is as tool to work with geospatial data in conjunction with dedicated geospatial applications and libraries and the module can not and should not replicate their functionality.

The only reimplemented feature is the most common WSG84 EPSG:3395 World Mercator projection, for everything else use the dedicated packages like:

- pyproj - Cartographic projections and coordinate transformations library.
- Shapely - Manipulation and analysis of geometric objects in the Cartesian plane.
- PyShp - The Python Shapefile Library (PyShp) reads and writes ESRI Shapefiles in pure Python.
- GeoJSON - GeoJSON interface for Python.
- GDAL - Tools for programming and manipulating the GDAL Geospatial Data Abstraction Library.
- Fiona - Fiona is GDAL's neat and nimble vector API for Python programmers.
- QGIS - A free and open source geographic information system.
- and many more …

This module provides support for the `__geo_interface__`: https://gist.github.com/sgillies/2217756

Which is also supported by Shapely, for supported types see the GeoJSON Standard and examples in Appendix-A.

See also:

Tutorial for the Geo Add-on for loading GPX data into DXF files with an existing geo location reference and exporting DXF entities as GeoJSON data.

Proxy From Mapping

The *GeoProxy* represents a `__geo_interface__` mapping, create a new proxy by *GeoProxy.parse()* from an external `__geo_interface__` mapping. *GeoProxy.to_dxf_entities()* returns new DXF entities from this mapping. Returns "Point" as *Point* entity, "LineString" as *LWPolyline* entity and "Polygon" as *Hatch* entity or as separated *LWPolyline* entities (or both) and new in v0.16.6 as *MPolygon*. Supports "MultiPoint", "MultiLineString", "MultiPolygon", "GeometryCollection", "Feature" and "FeatureCollection". Add new DXF entities to a layout by the `Layout.add_entity()` method.

Proxy From DXF Entity

The *proxy()* function or the constructor *GeoProxy.from_dxf_entities()* creates a new *GeoProxy* object from a single DXF entity or from an iterable of DXF entities, entities without a corresponding representation will be approximated.

Supported DXF entities are:

- POINT as "Point"
- LINE as "LineString"
- LWPOLYLINE as "LineString" if open and "Polygon" if closed
- POLYLINE as "LineString" if open and "Polygon" if closed, supports only 2D and 3D polylines, POLYMESH and POLYFACE are not supported
- SOLID, TRACE, 3DFACE as "Polygon"

- CIRCLE, ARC, ELLIPSE and SPLINE by approximation as "LineString" if open and "Polygon" if closed
- HATCH and MPOLYGON as "Polygon", holes are supported

Warning: This module does no extensive validity checks for "Polygon" objects and because DXF has different requirements for HATCH boundary paths than the GeoJSON Standard, it is possible to create invalid "Polygon" objects. It is recommended to check critical objects by a sophisticated geometry library like Shapely.

Module Functions

`ezdxf.addons.geo.`**`proxy`**(*entity: DXFGraphic | Iterable[DXFGraphic]*, *distance: float = MAX_FLATTENING_DISTANCE*, *force_line_string: bool = False*) → *GeoProxy*

Returns a *GeoProxy* object.

Parameters

- **entity** – a single DXF entity or iterable of DXF entities
- **distance** – maximum flattening distance for curve approximations
- **force_line_string** – by default this function returns Polygon objects for closed geometries like CIRCLE, SOLID, closed POLYLINE and so on, by setting argument *force_line_string* to `True`, this entities will be returned as LineString objects.

`ezdxf.addons.geo.`**`dxf_entities`**(*geo_mapping*, *polygon: int = 1*, *dxfattribs=None*) → Iterable[DXFGraphic]

Returns `__geo_interface__` mappings as DXF entities.

The *polygon* argument determines the method to convert polygons, use 1 for *Hatch* entity, 2 for *LWPolyline* or 3 for both. Option 2 returns for the exterior path and each hole a separated `LWPolyline` entity. The `Hatch` entity supports holes, but has no explicit border line.

Yields `Hatch` always before `LWPolyline` entities.

MPolygon support was added in v0.16.6, which is like a *Hatch* entity with additional border lines, but the MPOLYGON entity is not a core DXF entity and DXF viewers, applications and libraries my not support this entity. The DXF attribute *color* defines the border line color and *fill_color* the color of the solid filling.

The returned DXF entities can be added to a layout by the `Layout.add_entity()` method.

Parameters

- **geo_mapping** – `__geo__interface__` mapping as `dict` or a Python object with a `__geo__interface__` property
- **polygon** – method to convert polygons (1-2-3-4)
- **dxfattribs** – dict with additional DXF attributes

`ezdxf.addons.geo.`**`gfilter`**(*entities: Iterable[DXFGraphic]*) → Iterable[DXFGraphic]

Filter DXF entities from iterable *entities*, which are incompatible to the `__geo_reference__` interface.

GeoProxy Class

class ezdxf.addons.geo.**GeoProxy**(*geo_mapping: dict*, *places: int = 6*)

Stores the __geo_interface__ mapping in a parsed and compiled form.

Stores coordinates as `Vec3` objects and represents "Polygon" always as tuple (exterior, holes) even without holes.

The GeoJSON specification recommends 6 decimal places for latitude and longitude which equates to roughly 10cm of precision. You may need slightly more for certain applications, 9 decimal places would be sufficient for professional survey-grade GPS coordinates.

Parameters

- **geo_mapping** – parsed and compiled __geo_interface__ mapping
- **places** – decimal places to round for __geo_interface__ export

__geo_interface__

Returns the __geo_interface__ compatible mapping as `dict`.

geotype

Property returns the top level entity type or `None`.

classmethod parse(*geo_mapping: Dict*) → *GeoProxy*

Parse and compile a __geo_interface__ mapping as `dict` or a Python object with a __geo_interface__ property, does some basic syntax checks, converts all coordinates into `Vec3` objects, represents "Polygon" always as tuple (exterior, holes) even without holes.

classmethod from_dxf_entities(*entity: DXFGraphic | Iterable[DXFGraphic]*, *distance: float = MAX_FLATTENING_DISTANCE*, *force_line_string: bool = False*) → *GeoProxy*

Constructor from a single DXF entity or an iterable of DXF entities.

Parameters

- **entity** – DXF entity or entities
- **distance** – maximum flattening distance for curve approximations
- **force_line_string** – by default this function returns Polygon objects for closed geometries like CIRCLE, SOLID, closed POLYLINE and so on, by setting argument *force_line_string* to `True`, this entities will be returned as LineString objects.

to_dxf_entities(*polygon: int = 1*, *dxfattribs=None*) → Iterable[DXFGraphic]

Returns stored __geo_interface__ mappings as DXF entities.

The *polygon* argument determines the method to convert polygons, use 1 for *Hatch* entity, 2 for *LWPolyline* or 3 for both. Option 2 returns for the exterior path and each hole a separated `LWPolyline` entity. The `Hatch` entity supports holes, but has no explicit border line.

Yields `Hatch` always before `LWPolyline` entities.

MPolygon support was added in v0.16.6, which is like a *Hatch* entity with additional border lines, but the MPOLYGON entity is not a core DXF entity and DXF viewers, applications and libraries my not support this entity. The DXF attribute *color* defines the border line color and *fill_color* the color of the solid filling.

The returned DXF entities can be added to a layout by the `Layout.add_entity()` method.

Parameters

- **polygon** – method to convert polygons (1-2-3-4)
- **dxfattribs** – dict with additional DXF attributes

copy() → *GeoProxy*

Returns a deep copy.

__iter__() → Iterable[dict]

Iterate over all geo content objects.

Yields only "Point", "LineString", "Polygon", "MultiPoint", "MultiLineString" and "MultiPolygon" objects, returns the content of "GeometryCollection", "FeatureCollection" and "Feature" as geometry objects ("Point", ...).

wcs_to_crs(*crs:* Matrix44) → None

Transform all coordinates recursive from *WCS* coordinates into Coordinate Reference System (CRS) by transformation matrix *crs* inplace.

The CRS is defined by the *GeoData* entity, get the GeoData entity from the modelspace by method *get_geodata()*. The CRS transformation matrix can be acquired form the GeoData object by *get_crs_transformation()* method:

```
doc = ezdxf.readfile('file.dxf')
msp = doc.modelspace()
geodata = msp.get_geodata()
if geodata:
    matrix, axis_ordering = geodata.get_crs_transformation()
```

If *axis_ordering* is False the CRS is not compatible with the __geo_interface__ or GeoJSON (see chapter 3.1.1).

Parameters
crs – transformation matrix of type *Matrix44*

crs_to_wcs(*crs:* Matrix44) → None

Transform all coordinates recursive from CRS into *WCS* coordinates by transformation matrix *crs* inplace, see also *GeoProxy.wcs_to_crs()*.

Parameters
crs – transformation matrix of type *Matrix44*

globe_to_map(*func: Callable[[*Vec3*], *Vec3*]* | *None* = *None*) → None

Transform all coordinates recursive from globe representation in longitude and latitude in decimal degrees into 2D map representation in meters.

Default is WGS84 EPSG:4326 (GPS) to WGS84 EPSG:3395 World Mercator function *wgs84_4326_to_3395()*.

Use the pyproj package to write a custom projection function as needed.

Parameters
func – custom transformation function, which takes one Vec3 object as argument and returns the result as a Vec3 object.

map_to_globe(*func: Callable[[*Vec3*], *Vec3*]* | *None* = *None*) → None

Transform all coordinates recursive from 2D map representation in meters into globe representation as longitude and latitude in decimal degrees.

Default is WGS84 EPSG:3395 World Mercator to WGS84 EPSG:4326 GPS function *wgs84_3395_to_4326()*.

Use the pyproj package to write a custom projection function as needed.

Parameters
func – custom transformation function, which takes one `Vec3` object as argument and returns the result as a `Vec3` object.

apply(*func: Callable[[Vec3], Vec3]*) → None

Apply the transformation function *func* recursive to all coordinates.

Parameters
func – transformation function as Callable[[Vec3], Vec3]

filter(*func: Callable[[GeoProxy], bool]*) → None

Removes all mappings for which *func()* returns `False`. The function only has to handle Point, LineString and Polygon entities, other entities like MultiPolygon are divided into separate entities also any collection.

Helper Functions

ezdxf.addons.geo.**wgs84_4326_to_3395**(*location:* Vec3) → *Vec3*

Transform WGS84 EPSG:4326 location given as latitude and longitude in decimal degrees as used by GPS into World Mercator cartesian 2D coordinates in meters EPSG:3395.

Parameters
location – `Vec3` object, x-attribute represents the longitude value (East-West) in decimal degrees and the y-attribute represents the latitude value (North-South) in decimal degrees.

ezdxf.addons.geo.**wgs84_3395_to_4326**(*location:* Vec3, *tol: float = 1e-6*) → *Vec3*

Transform WGS84 World Mercator EPSG:3395 location given as cartesian 2D coordinates x, y in meters into WGS84 decimal degrees as longitude and latitude EPSG:4326 as used by GPS.

Parameters

- **location** – `Vec3` object, z-axis is ignored
- **tol** – accuracy for latitude calculation

ezdxf.addons.geo.**dms2dd**(*d: float*, *m: float = 0*, *s: float = 0*) → float

Convert degree, minutes, seconds into decimal degrees.

ezdxf.addons.geo.**dd2dms**(*dd: float*) → tuple[float, float, float]

Convert decimal degrees into degree, minutes, seconds.

6.12.3 Importer

This add-on is meant to import graphical entities from another DXF drawing and their required table entries like LAYER, LTYPE or STYLE.

Because of complex extensibility of the DXF format and the lack of sufficient documentation, I decided to remove most of the possible source drawing dependencies from imported entities, therefore imported entities may not look the same as the original entities in the source drawing, but at least the geometry should be the same and the DXF file does not break.

Removed data which could contain source drawing dependencies: Extension Dictionaries, AppData and XDATA.

Warning: DON'T EXPECT PERFECT RESULTS!

The *Importer* supports following data import:

- entities which are really safe to import: LINE, POINT, CIRCLE, ARC, TEXT, SOLID, TRACE, 3DFACE, SHAPE, POLYLINE, ATTRIB, ATTDEF, INSERT, ELLIPSE, MTEXT, LWPOLYLINE, SPLINE, HATCH, MESH, XLINE, RAY, DIMENSION, LEADER, VIEWPORT
- table and table entry import is restricted to LAYER, LTYPE, STYLE, DIMSTYLE
- import of BLOCK definitions is supported
- import of paper space layouts is supported

Import of DXF objects from the OBJECTS section is not supported.

DIMSTYLE override for entities DIMENSION and LEADER is not supported.

Example:

```
import ezdxf
from ezdxf.addons import Importer

sdoc = ezdxf.readfile('original.dxf')
tdoc = ezdxf.new()

importer = Importer(sdoc, tdoc)

# import all entities from source modelspace into modelspace of the target drawing
importer.import_modelspace()

# import all paperspace layouts from source drawing
importer.import_paperspace_layouts()

# import all CIRCLE and LINE entities from source modelspace into an arbitrary target␣
↪layout.
# create target layout
tblock = tdoc.blocks.new('SOURCE_ENTS')
# query source entities
ents = sdoc.modelspace().query('CIRCLE LINE')
# import source entities into target block
importer.import_entities(ents, tblock)

# This is ALWAYS the last & required step, without finalizing the target drawing is␣
↪maybe invalid!
# This step imports all additional required table entries and block definitions.
importer.finalize()

tdoc.saveas('imported.dxf')
```

class ezdxf.addons.importer.**Importer**(*source:* Drawing, *target:* Drawing)

The *Importer* class is central element for importing data from other DXF documents.

Parameters

- **source** – source Drawing
- **target** – target Drawing

source

source DXF document

target

target DXF document

used_layers

Set of used layer names as string, AutoCAD accepts layer names without a LAYER table entry.

used_linetypes

Set of used linetype names as string, these linetypes require a TABLE entry or AutoCAD will crash.

used_styles

Set of used text style names, these text styles require a TABLE entry or AutoCAD will crash.

used_dimstyles

Set of used dimension style names, these dimension styles require a TABLE entry or AutoCAD will crash.

finalize() → None

Finalize the import by importing required table entries and BLOCK definitions, without finalization the target document is maybe invalid for AutoCAD. Call the *finalize()* method as last step of the import process.

import_block(*block_name: str*, *rename=True*) → str

Import one BLOCK definition from source document.

If the BLOCK already exist the BLOCK will be renamed if argument *rename* is `True`, otherwise the existing BLOCK in the target document will be used instead of the BLOCK in the source document. Required name resolving for imported block references (INSERT), will be done in the *Importer.finalize()* method.

To replace an existing BLOCK in the target document, just delete it before importing data: `target.blocks.delete_block(block_name, safe=False)`

Parameters

- **block_name** – name of BLOCK to import
- **rename** – rename BLOCK if a BLOCK with the same name already exist in target document

Returns: (renamed) BLOCK name

Raises

ValueError – BLOCK in source document not found (defined)

import_blocks(*block_names: Iterable[str]*, *rename=False*) → None

Import all BLOCK definitions from source document.

If a BLOCK already exist the BLOCK will be renamed if argument *rename* is `True`, otherwise the existing BLOCK in the target document will be used instead of the BLOCK from the source document. Required name resolving for imported BLOCK references (INSERT), will be done in the *Importer.finalize()* method.

Parameters

- **block_names** – names of BLOCK definitions to import
- **rename** – rename BLOCK if a BLOCK with the same name already exist in target document

Raises

ValueError – BLOCK in source document not found (defined)

import_entities(*entities: Iterable[DXFEntity]*, *target_layout:* BaseLayout | *None* = *None*) → None

Import all *entities* into *target_layout* or the modelspace of the target document, if *target_layout* is `None`.

Parameters

- **entities** – Iterable of DXF entities
- **target_layout** – any layout (modelspace, paperspace or block) from the target document

Raises
DXFStructureError – *target_layout* is not a layout of target document

import_entity(*entity:* DXFEntity, *target_layout:* BaseLayout | *None* = *None*) → None

Imports a single DXF *entity* into *target_layout* or the modelspace of the target document, if *target_layout* is `None`.

Parameters

- **entity** – DXF entity to import
- **target_layout** – any layout (modelspace, paperspace or block) from the target document

Raises
DXFStructureError – *target_layout* is not a layout of target document

import_modelspace(*target_layout:* BaseLayout | *None* = *None*) → None

Import all entities from source modelspace into *target_layout* or the modelspace of the target document, if *target_layout* is `None`.

Parameters
target_layout – any layout (modelspace, paperspace or block) from the target document

Raises
DXFStructureError – *target_layout* is not a layout of target document

import_paperspace_layout(*name: str*) → *Layout*

Import paperspace layout *name* into the target document.

Recreates the source paperspace layout in the target document, renames the target paperspace if a paperspace with same *name* already exist and imports all entities from the source paperspace into the target paperspace.

Parameters
name – source paper space name as string

Returns: new created target paperspace `Layout`

Raises

- **KeyError** – source paperspace does not exist
- ***DXFTypeError*** – invalid modelspace import

import_paperspace_layouts() → None

Import all paperspace layouts and their content into the target document. Target layouts will be renamed if a layout with the same name already exist. Layouts will be imported in original tab order.

import_shape_files(*fonts: set[str]*) → None

Import shape file table entries from the source document into the target document. Shape file entries are stored in the styles table but without a name.

import_table(*name: str*, *entries: str | Iterable[str] = '*'*, *replace=False*) → None

Import specific table entries from the source document into the target document.

Parameters

- **name** – valid table names are “layers”, “linetypes” and “styles”
- **entries** – Iterable of table names as strings, or a single table name or “*” for all table entries
- **replace** – `True` to replace the already existing table entry else ignore existing entries

Raises
TypeError – unsupported table type

import_tables (*table_names: str | Iterable[str] = '*', replace=False*) → None
Import DXF tables from the source document into the target document.

Parameters

- **table_names** – iterable of tables names as strings, or a single table name as string or "*" for all supported tables
- **replace** – `True` to replace already existing table entries else ignore existing entries

Raises
TypeError – unsupported table type

recreate_source_layout (*name: str*) → *Layout*
Recreate source paperspace layout *name* in the target document. The layout will be renamed if *name* already exist in the target document. Returns target modelspace for layout name "Model".

Parameters
name – layout name as string

Raises
KeyError – if source layout *name* not exist

6.12.4 dxf2code

Translate DXF entities and structures into Python source code.

Short example:

```
import ezdxf
from ezdxf.addons.dxf2code import entities_to_code, block_to_code

doc = ezdxf.readfile('original.dxf')
msp = doc.modelspace()
source = entities_to_code(msp)

# create source code for a block definition
block_source = block_to_code(doc.blocks['MyBlock'])

# merge source code objects
source.merge(block_source)

with open('source.py', mode='wt') as f:
    f.write(source.import_str())
    f.write('\n\n')
    f.write(source.code_str())
    f.write('\n')
```

ezdxf.addons.dxf2code.**entities_to_code** (*entities: Iterable[DXFEntity], layout: str = 'layout', ignore: Iterable[str] | None = None*) → *Code*
Translates DXF entities into Python source code to recreate this entities by ezdxf.

Parameters

- **entities** – iterable of DXFEntity
- **layout** – variable name of the layout (model space or block) as string

- **ignore** – iterable of entities types to ignore as strings like `['IMAGE', 'DIMENSION']`

Returns
Code

`ezdxf.addons.dxf2code.`**`block_to_code`**(*block:* BlockLayout, *drawing: str = 'doc'*, *ignore: Iterable[str] | None = None*) → *Code*

Translates a BLOCK into Python source code to recreate the BLOCK by ezdxf.

Parameters

- **block** – block definition layout
- **drawing** – variable name of the drawing as string
- **ignore** – iterable of entities types to ignore as strings like ['IMAGE', 'DIMENSION']

Returns
Code

`ezdxf.addons.dxf2code.`**`table_entries_to_code`**(*entities: Iterable[*DXFEntity*]*, *drawing='doc'*) → *Code*

`ezdxf.addons.dxf2code.`**`black`**(*code: str*, *line_length=88*, *fast: bool = True*) → str

Returns the source *code* as a single string formatted by Black

Requires the installed Black formatter:

```
pip3 install black
```

Parameters

- **code** – source code
- **line_length** – max. source code line length
- **fast** – `True` for fast mode, `False` to check that the reformatted code is valid

Raises
ImportError – Black is not available

class `ezdxf.addons.dxf2code.`**`Code`**

Source code container.

code
Source code line storage, store lines without line ending `\\n`

imports
source code line storage for global imports, store lines without line ending `\\n`

layers
Layers used by the generated source code, AutoCAD accepts layer names without a LAYER table entry.

linetypes
Linetypes used by the generated source code, these linetypes require a TABLE entry or AutoCAD will crash.

styles
Text styles used by the generated source code, these text styles require a TABLE entry or AutoCAD will crash.

dimstyles

Dimension styles used by the generated source code, these dimension styles require a TABLE entry or AutoCAD will crash.

blocks

Blocks used by the generated source code, these blocks require a BLOCK definition in the BLOCKS section or AutoCAD will crash.

code_str(*indent: int = 0*) → str

Returns the source code as a single string.

Parameters
indent – source code indentation count by spaces

black_code_str(*line_length=88*) → str

Returns the source code as a single string formatted by Black

Parameters
line_length – max. source code line length

Raises
ImportError – Black is not available

import_str(*indent: int = 0*) → str

Returns required imports as a single string.

Parameters
indent – source code indentation count by spaces

merge(*code:* Code, *indent: int = 0*) → None

Add another *Code* object.

add_import(*statement: str*) → None

Add import statement, identical import statements are merged together.

add_line(*code: str*, *indent: int = 0*) → None

Add a single source code line without line ending `\n`.

add_lines(*code: Iterable[str]*, *indent: int = 0*) → None

Add multiple source code lines without line ending `\n`.

6.12.5 iterdxf

This add-on allows iterating over entities of the modelspace of really big (> 5GB) DXF files which do not fit into memory by only loading one entity at the time. Only ASCII DXF files are supported.

The entities are regular *DXFGraphic* objects with access to all supported DXF attributes, this entities can be written to new DXF files created by the *IterDXF.export()* method. The new *add_foreign_entity()* method allows also to add this entities to new regular *ezdxf* drawings (except for the INSERT entity), but resources like linetype and style are removed, only layer will be preserved but only with default attributes like color `7` and linetype `CONTINUOUS`.

The following example shows how to split a big DXF files into several separated DXF files which contains only LINE, TEXT or POLYLINE entities.

```
from ezdxf.addons import iterdxf

doc = iterdxf.opendxf('big.dxf')
line_exporter = doc.export('line.dxf')
```

(continues on next page)

(continued from previous page)

```
text_exporter = doc.export('text.dxf')
polyline_exporter = doc.export('polyline.dxf')
try:
    for entity in doc.modelspace():
        if entity.dxftype() == 'LINE':
            line_exporter.write(entity)
        elif entity.dxftype() == 'TEXT':
            text_exporter.write(entity)
        elif entity.dxftype() == 'POLYLINE':
            polyline_exporter.write(entity)
finally:
    line_exporter.close()
    text_exporter.close()
    polyline_exporter.close()
    doc.close()
```

Supported DXF types:

3DFACE, ARC, ATTDEF, ATTRIB, CIRCLE, DIMENSION, ELLIPSE, HATCH, HELIX, IMAGE, INSERT, LEADER, LINE, LWPOLYLINE, MESH, MLEADER, MLINE, MTEXT, POINT, POLYLINE, RAY, SHAPE, SOLID, SPLINE, TEXT, TRACE, VERTEX, WIPEOUT, XLINE

Transfer simple entities to another DXF document, this works for some supported entities, except for entities with strong dependencies to the original document like INSERT look at *add_foreign_entity()* for all supported types:

```
newdoc = ezdxf.new()
msp = newdoc.modelspace()
# line is an entity from a big source file
msp.add_foreign_entity(line)
# and so on ...
msp.add_foreign_entity(lwpolyline)
msp.add_foreign_entity(mesh)
msp.add_foreign_entity(polyface)
```

Transfer MESH and POLYFACE (dxftype for POLYFACE and POLYMESH is POLYLINE!) entities into a new DXF document by the `MeshTransformer` class:

```
from ezdxf.render import MeshTransformer

# mesh is MESH from a big source file
t = MeshTransformer.from_mesh(mesh)
# create a new MESH entity from MeshTransformer
t.render(msp)

# polyface is POLYFACE from a big source file
t = MeshTransformer.from_polyface(polyface)
# create a new POLYMESH entity from MeshTransformer
t.render_polyface(msp)
```

Another way to import entities from a big source file into new DXF documents is to split the big file into smaller parts and use the *Importer* add-on for a more safe entity import.

`ezdxf.addons.iterdxf.`**`opendxf`**(*filename: Path | str*, *errors: str = 'surrogateescape'*) → *IterDXF*

Open DXF file for iterating, be sure to open valid DXF files, no DXF structure checks will be applied.

Use this function to split up big DXF files as shown in the example above.

Parameters

- **filename** – DXF filename of a seekable DXF file.
- **errors** – specify decoding error handler
  - ”surrogateescape” to preserve possible binary data (default)
  - ”ignore” to use the replacement char U+FFFD “” for invalid data
  - ”strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- ***DXFStructureError*** – invalid or incomplete DXF file
- **UnicodeDecodeError** – if *errors* is “strict” and a decoding error occurs

`ezdxf.addons.iterdxf.`**modelspace**(*filename: Path | str*, *types: Iterable[str] | None = None*, *errors: str = 'surrogateescape'*) → Iterable[DXFGraphic]

Iterate over all modelspace entities as `DXFGraphic` objects of a seekable file.

Use this function to iterate “quick” over modelspace entities of a DXF file, filtering DXF types may speed up things if many entity types will be skipped.

Parameters

- **filename** – filename of a seekable DXF file
- **types** – DXF types like `['LINE', '3DFACE']` which should be returned, `None` returns all supported types.
- **errors** – specify decoding error handler
  - ”surrogateescape” to preserve possible binary data (default)
  - ”ignore” to use the replacement char U+FFFD “” for invalid data
  - ”strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- ***DXFStructureError*** – invalid or incomplete DXF file
- **UnicodeDecodeError** – if *errors* is “strict” and a decoding error occurs

`ezdxf.addons.iterdxf.`**single_pass_modelspace**(*stream: BinaryIO*, *types: Iterable[str] | None = None*, *errors: str = 'surrogateescape'*) → Iterable[DXFGraphic]

Iterate over all modelspace entities as `DXFGraphic` objects in a single pass.

Use this function to ‘quick’ iterate over modelspace entities of a **not** seekable binary DXF stream, filtering DXF types may speed up things if many entity types will be skipped.

Parameters

- **stream** – (not seekable) binary DXF stream
- **types** – DXF types like `['LINE', '3DFACE']` which should be returned, `None` returns all supported types.
- **errors** – specify decoding error handler
  - ”surrogateescape” to preserve possible binary data (default)
  - ”ignore” to use the replacement char U+FFFD “” for invalid data
  - ”strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- ***DXFStructureError*** – Invalid or incomplete DXF file
- **UnicodeDecodeError** – if *errors* is "strict" and a decoding error occurs

class ezdxf.addons.iterdxf.**IterDXF**

export(*name: Path | str*) → *IterDXFWriter*

Returns a companion object to export parts from the source DXF file into another DXF file, the new file will have the same HEADER, CLASSES, TABLES, BLOCKS and OBJECTS sections, which guarantees all necessary dependencies are present in the new file.

Parameters
name – filename, no special requirements

modelspace(*types: Iterable[str] | None = None*) → Iterable[DXFGraphic]

Returns an iterator for all supported DXF entities in the modelspace. These entities are regular *DXFGraphic* objects but without a valid document assigned. It is **not** possible to add these entities to other *ezdxf* documents.

It is only possible to recreate the objects by factory functions base on attributes of the source entity. For MESH, POLYMESH and POLYFACE it is possible to use the *MeshTransformer* class to render (recreate) this objects as new entities in another document.

Parameters
types – DXF types like `['LINE', '3DFACE']` which should be returned, `None` returns all supported types.

close()

Safe closing source DXF file.

class ezdxf.addons.iterdxf.**IterDXFWriter**

write(*entity: DXFGraphic*)

Write a DXF entity from the source DXF file to the export file.

Don't write entities from different documents than the source DXF file, dependencies and resources will not match, maybe it will work once, but not in a reliable way for different DXF documents.

close()

Safe closing of exported DXF file. Copying of OBJECTS section happens only at closing the file, without closing the new DXF file is invalid.

6.12.6 ODA File Converter Support

Use an installed ODA File Converter for converting between different versions of *.dwg*, *.dxb* and *.dxf*.

Warning: Execution of an external application is a big security issue! Especially when the path to the executable can be altered.

To avoid this problem delete the `ezdxf.addons.odafc.py` module.

Install ODA File Converter

The ODA File Converter has to be installed by the user, the application is available for Windows XP, Windows 7 or later, Mac OS X, and Linux in 32/64-bit RPM and DEB format.

AppImage Support

The option "unix_exec_path" defines an executable for Linux and macOS, this executable overrides the default command `ODAFileConverter`. Assign an **absolute** path to the executable to that key and if the executable is not found the add-on falls back to the `ODAFileConverter` command.

The option "unix_exec_path" also adds support for AppImages provided by the Open Design Alliance. Download the AppImage file and store it in a folder of your choice (e.g. `~/Apps`) and make the file executable:

```
chmod a+x ~/Apps/ODAFileConverter_QT5_lnxX64_8.3dll_23.9.AppImage
```

Add the **absolute** path as config option "unix_exec_path" to the "odafc-addon" section:

```
[odafc-addon]
win_exec_path = "C:\Program Files\ODA\ODAFileConverter\ODAFileConverter.exe"
unix_exec_path = "/home/<your user name>/Apps/ODAFileConverter_QT5_lnxX64_8.3dll_23.9.
↪AppImage"
```

This overrides the default command `ODAFileConverter` and if the executable is not found the add-on falls back to the `ODAFileConverter` command.

See also:

For more information about config files see section: *Global Options Object*

Suppressed GUI

On Windows the GUI of the ODA File Converter is suppressed, on Linux you may have to install the `xvfb` package to prevent this, for macOS is no solution known.

Supported DXF and DWG Versions

ODA File Converter version strings, you can use any of this strings to specify a version, `'R..'` and `'AC....'` strings will be automatically mapped to `'ACAD....'` strings:

ODAFC	ezdxf	Version
ACAD9	not supported	AC1004
ACAD10	not supported	AC1006
ACAD12	R12	AC1009
ACAD13	R13	AC1012
ACAD14	R14	AC1014
ACAD2000	R2000	AC1015
ACAD2004	R2004	AC1018
ACAD2007	R2007	AC1021
ACAD2010	R2010	AC1024
ACAD2013	R2013	AC1027
ACAD2018	R2018	AC1032

Config

On Windows the path to the `ODAFileConverter.exe` executable is stored in the config file (see *ezdxf.options*) in the "odafc-addon" section as key "win_exec_path", the default entry is:

```
[odafc-addon]
win_exec_path = "C:\Program Files\ODA\ODAFileConverter\ODAFileConverter.exe"
unix_exec_path =
```

On Linux and macOS the `ODAFileConverter` command is located by the `shutil.which()` function but can be overridden since version 1.0 by the key "linux_exec_path".

Usage

```
from ezdxf.addons import odafc

# Load a DWG file
doc = odafc.readfile('my.dwg')

# Use loaded document like any other ezdxf document
print(f'Document loaded as DXF version: {doc.dxfversion}.')
msp = doc.modelspace()
...

# Export document as DWG file for AutoCAD R2018
odafc.export_dwg(doc, 'my_R2018.dwg', version='R2018')
```

`ezdxf.addons.odafc.`**`win_exec_path`**

Path to installed *ODA File Converter* executable on Windows systems, default is `"C:\Program Files\ODA\ODAFileConverter\ODAFileConverter.exe"`.

`ezdxf.addons.odafc.`**`unix_exec_path`**

Absolute path to a Linux or macOS executable if set, otherwise an empty string and the default command `ODAFileConverter` is used.

`ezdxf.addons.odafc.`**`is_installed`**`()` → bool

Returns `True` if the ODAFileConverter is installed.

`ezdxf.addons.odafc.`**`readfile`**(*filename: str*, *version: str | None = None*, *, *audit: bool = False*) → *Drawing* | None

Uses an installed ODA File Converter to convert a DWG/DXB/DXF file into a temporary DXF file and load this file by *ezdxf*.

Parameters

- **`filename`** – file to load by ODA File Converter
- **`version`** – load file as specific DXF version, by default the same version as the source file or if not detectable the latest by *ezdxf* supported version.
- **`audit`** – audit source file before loading

Raises

- **`FileNotFoundError`** – source file not found
- **`odafc.UnknownODAFCError`** – conversion failed for unknown reasons
- **`odafc.UnsupportedVersion`** – invalid DWG version specified

- **odafc.UnsupportedFileFormat** – unsupported file extension
- **odafc.ODAFCNotInstalledError** – ODA File Converter not installed

ezdxf.addons.odafc.**export_dwg**(*doc: Drawing*, *filename: str*, *version: str | None = None*, *, *audit: bool = False*, *replace: bool = False*) → None

Uses an installed ODA File Converter to export the DXF document *doc* as a DWG file.

A temporary DXF file will be created and converted to DWG by the ODA File Converter. If *version* is not specified the DXF version of the source document is used.

Parameters

- **doc** – *ezdxf* DXF document as `Drawing` object
- **filename** – output DWG filename, the extension will be set to ".dwg"
- **version** – DWG version to export, by default the same version as the source document.
- **audit** – audit source file by ODA File Converter at exporting
- **replace** – replace existing DWG file if `True`

Raises

- **FileExistsError** – target file already exists, and argument *replace* is `False`
- **FileNotFoundError** – parent directory of target file does not exist
- **odafc.UnknownODAFCError** – exporting DWG failed for unknown reasons
- **odafc.ODAFCNotInstalledError** – ODA File Converter not installed

ezdxf.addons.odafc.**convert**(*source: str | Path*, *dest: str | Path = ''*, *, *version='R2018'*, *audit=True*, *replace=False*)

Convert *source* file to *dest* file.

The file extension defines the target format e.g. `convert("test.dxf", "Test.dwg")` converts the source file to a DWG file. If *dest* is an empty string the conversion depends on the source file format and is DXF to DWG or DWG to DXF. To convert DXF to DXF an explicit destination filename is required: `convert("r12.dxf", "r2013.dxf", version="R2013")`

Parameters

- **source** – source file
- **dest** – destination file, an empty string uses the source filename with the extension of the target format e.g. "test.dxf" -> "test.dwg"
- **version** – output DXF/DWG version e.g. "ACAD2018", "R2018", "AC1032"
- **audit** – audit files
- **replace** – replace existing destination file

Raises

- **FileNotFoundError** – source file or destination folder does not exist
- **FileExistsError** – destination file already exists and argument *replace* is `False`
- **odafc.UnsupportedVersion** – invalid DXF version specified
- **odafc.UnsupportedFileFormat** – unsupported file extension
- **odafc.UnknownODAFCError** – conversion failed for unknown reasons
- **odafc.ODAFCNotInstalledError** – ODA File Converter not installed

6.12.7 R12 Export

New in version 1.1.

This module exports any DXF file as a simple DXF R12 file. Many complex entities will be converted into DXF primitives. This exporter is intended for creating a simple file format as an input format for other software such as laser cutters. In order to get a file that can be edited well in a CAD application, the results of the ODA file converter are much better.

Usage

```
import ezdxf
from ezdxf.addons import r12export

doc = ezdxf.readfile("any.dxf")
r12export.saveas(doc, "r12.dxf")
```

Converted Entity Types

LWPOLYLINE	translated to POLYLINE
MESH	translated to POLYLINE (PolyfaceMesh)
SPLINE	flattened to POLYLINE
ELLIPSE	flattened to POLYLINE
MTEXT	exploded into DXF primitives
LEADER	exploded into DXF primitives
MLEADER	exploded into DXF primitives
MULTILEADER	exploded into DXF primitives
MLINE	exploded into DXF primitives
HATCH	exploded into DXF primitives
MPOLYGON	exploded into DXF primitives
ACAD_TABLE	export of pre-rendered BLOCK content

For proxy- or unknown entities the available proxy graphic will be exported as DXF primitives.

Limitations

- Explosion of MTEXT into DXF primitives is not perfect
- Pattern rendering for complex HATCH entities has issues
- Solid fill rendering for complex HATCH entities has issues

ODA File Converter

The advantage of the *r12export* module is that the ODA file converter isn't needed, but the ODA file converter will produce a much better result:

```
from ezdxf.addons import odafc

odafc.convert("any.dxf", "r12.dxf", version="R12")
```

Functions

write	Write a DXF document as DXF version R12 to a text stream.
saveas	Write a DXF document as DXF version R12 to a file.
convert	Export and reload DXF document as DXF version R12.

ezdxf.addons.r12export.**write**(*doc:* Drawing, *stream: TextIO*, *, *max_sagitta: float = MAX_SAGITTA*) → None

Write a DXF document as DXF version R12 to a text stream. The *max_sagitta* argument determines the accuracy of the curve flatting for SPLINE and ELLIPSE entities.

Parameters

- **doc** – DXF document to export
- **stream** – output stream, use `doc.encoding` as encoding
- **max_sagitta** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.

ezdxf.addons.r12export.**saveas**(*doc:* Drawing, *filepath: str | PathLike*, *, *max_sagitta: float = MAX_SAGITTA*) → None

Write a DXF document as DXF version R12 to a file. The *max_sagitta* argument determines the accuracy of the curve flatting for SPLINE and ELLIPSE entities.

Parameters

- **doc** – DXF document to export
- **filepath** – output filename
- **max_sagitta** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.

ezdxf.addons.r12export.**convert**(*doc:* Drawing, *, *max_sagitta: float = MAX_SAGITTA*) → *Drawing*

Export and reload DXF document as DXF version R12.

Writes the DXF document into a temporary file at the file-system and reloads this file by the *ezdxf.readfile()* function.

6.12.8 r12writer

The fast file/stream writer creates simple DXF R12 drawings with just an ENTITIES section. The HEADER, TABLES and BLOCKS sections are not present except FIXED-TABLES are written. Only LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE entities are supported. FIXED-TABLES is a predefined TABLES section, which will be written, if the init argument *fixed_tables* of *R12FastStreamWriter* is `True`.

The *R12FastStreamWriter* writes the DXF entities as strings direct to the stream without creating an in-memory drawing and therefore the processing is very fast.

Because of the lack of a BLOCKS section, BLOCK/INSERT can not be used. Layers can be used, but this layers have a default setting color = 7 (black/white) and linetype = `'Continuous'`. If writing the FIXED-TABLES, some predefined text styles and line types are available, else text style is always `'STANDARD'` and line type is always `'ByLayer'`.

If using FIXED-TABLES, following predefined line types are available:

- CONTINUOUS
- CENTER ____ _ ____ _ ____ _ ____ _ ____ _ ____
- CENTERX2 __________ __ __________ __ __________
- CENTER2 ____ _ ____ _ ____ _ ____ _ ____
- DASHED __ __ __ __ __ __ __ __ __ __ __ __ __ _
- DASHEDX2 ____ ____ ____ ____ ____ ____
- DASHED2 _ _ _ _ _ _ _ _ _ _ _ _ _ _
- PHANTOM _______ __ __ _______ __ __ _______
- PHANTOMX2 ______________ ____ ____ ______________
- PHANTOM2 ___ _ _ ___ _ _ ___ _ _ ___ _ _ ___
- DASHDOT __ . __ . __ . __ . __ . __ . __ . __
- DASHDOTX2 ____ . ____ . ____ . ____
- DASHDOT2 _ . _ . _ . _ . _ . _ . _ . _ . _
- DOT . . . . . .
- DOTX2 . . . . . .
- DOT2 . . . . . .
- DIVIDE __ . . __ . . __ . . __ . . __ . . __
- DIVIDEX2 ____ . . ____ . . ____ . . ____
- DIVIDE2 _ . _ . _ . _ . _ . _ . _ . _ . _

If using FIXED-TABLES, following predefined text styles are available:

- OpenSans
- OpenSansCondensed-Light

Tutorial

A simple example with different DXF entities:

```
from random import random
from ezdxf.addons import r12writer

with r12writer("quick_and_dirty_dxf_r12.dxf") as dxf:
    dxf.add_line((0, 0), (17, 23))
    dxf.add_circle((0, 0), radius=2)
    dxf.add_arc((0, 0), radius=3, start=0, end=175)
    dxf.add_solid([(0, 0), (1, 0), (0, 1), (1, 1)])
    dxf.add_point((1.5, 1.5))

    # 2d polyline, new in v0.12
    dxf.add_polyline_2d([(5, 5), (7, 3), (7, 6)])

    # 2d polyline with bulge value, new in v0.12
    dxf.add_polyline_2d([(5, 5), (7, 3, 0.5), (7, 6)], format='xyb')

    # 3d polyline only, changed in v0.12
    dxf.add_polyline([(4, 3, 2), (8, 5, 0), (2, 4, 9)])

    dxf.add_text("test the text entity", align="MIDDLE_CENTER")
```

A simple example of writing really many entities in a short time:

```
from random import random
from ezdxf.addons import r12writer

MAX_X_COORD = 1000.0
MAX_Y_COORD = 1000.0
CIRCLE_COUNT = 1000000

with r12writer("many_circles.dxf") as dxf:
    for i in range(CIRCLE_COUNT):
        dxf.add_circle((MAX_X_COORD*random(), MAX_Y_COORD*random()), radius=2)
```

Show all available line types:

```
import ezdxf

LINETYPES = [
    'CONTINUOUS', 'CENTER', 'CENTERX2', 'CENTER2',
    'DASHED', 'DASHEDX2', 'DASHED2', 'PHANTOM', 'PHANTOMX2',
    'PHANTOM2', 'DASHDOT', 'DASHDOTX2', 'DASHDOT2', 'DOT',
    'DOTX2', 'DOT2', 'DIVIDE', 'DIVIDEX2', 'DIVIDE2',
]

with r12writer('r12_linetypes.dxf', fixed_tables=True) as dxf:
    for n, ltype in enumerate(LINETYPES):
        dxf.add_line((0, n), (10, n), linetype=ltype)
        dxf.add_text(ltype, (0, n+0.1), height=0.25, style='OpenSansCondensed-Light')
```

Reference

ezdxf.addons.r12writer.**r12writer**(*stream: TextIO | BinaryIO | str*, *fixed_tables=False*, *fmt='asc'*) → *R12FastStreamWriter*

Context manager for writing DXF entities to a stream/file. *stream* can be any file like object with a `write()` method or just a string for writing DXF entities to the file system. If *fixed_tables* is `True`, a standard TABLES section is written in front of the ENTITIES section and some predefined text styles and line types can be used.

Set argument *fmt* to "asc" to write ASCII DXF file (default) or "bin" to write Binary DXF files. ASCII DXF require a `TextIO` stream and Binary DXF require a `BinaryIO` stream.

class ezdxf.addons.r12writer.**R12FastStreamWriter**(*stream: TextIO*, *fixed_tables=False*)

Fast stream writer to create simple DXF R12 drawings.

Parameters

- **stream** – a file like object with a `write()` method.
- **fixed_tables** – if *fixed_tables* is `True`, a standard TABLES section is written in front of the ENTITIES section and some predefined text styles and line types can be used.

close() → None

Writes the DXF tail. Call is not necessary when using the context manager *r12writer()*.

add_line(*start: Sequence[float]*, *end: Sequence[float]*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a LINE entity from *start* to *end*.

Parameters

- **start** – start vertex as `(x, y[, z])` tuple
- **end** – end vertex as as `(x, y[, z])` tuple
- **layer** – layer name as string, without a layer definition the assigned color = `7` (black/white) and line type is `'Continuous'`.
- **color** – color as *AutoCAD Color Index (ACI)* in the range from `0` to `256`, `0` is *ByBlock* and `256` is *ByLayer*, default is *ByLayer* which is always color = `7` (black/white) without a layer definition.
- **linetype** – line type as string, if FIXED-TABLES are written some predefined line types are available, else line type is always *ByLayer*, which is always `'Continuous'` without a LAYERS table.

add_circle(*center: Sequence[float]*, *radius: float*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a CIRCLE entity.

Parameters

- **center** – circle center point as `(x, y)` tuple
- **radius** – circle radius as float
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_arc(*center: Sequence[float]*, *radius: float*, *start: float = 0*, *end: float = 360*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add an ARC entity. The arc goes counter-clockwise from *start* angle to *end* angle.

Parameters

- **center** – arc center point as `(x, y)` tuple
- **radius** – arc radius as float
- **start** – arc start angle in degrees as float
- **end** – arc end angle in degrees as float
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_point(*location: Sequence[float]*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a POINT entity.

Parameters

- **location** – point location as `(x, y [,z])` tuple
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_3dface(*vertices: Iterable[Sequence[float]]*, *invisible: int = 0*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a 3DFACE entity. 3DFACE is a spatial area with 3 or 4 vertices, all vertices have to be in the same plane.

Parameters

- **vertices** – iterable of 3 or 4 `(x, y, z)` vertices.
- **invisible** – bit coded flag to define the invisible edges,
  1. edge = 1
  2. edge = 2
  3. edge = 4
  4. edge = 8

  Add edge values to set multiple edges invisible, 1. edge + 3. edge = 1 + 4 = 5, all edges = 15
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_solid(*vertices: Iterable[Sequence[float]]*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a SOLID entity. SOLID is a solid filled area with 3 or 4 edges and SOLID is a 2D entity.

Parameters

- **vertices** – iterable of 3 or 4 `(x, y[, z])` tuples, z-axis will be ignored.
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_polyline_2d(*points: Iterable[Sequence]*, *format: str = 'xy'*, *closed: bool = False*, *start_width: float = 0*, *end_width: float = 0*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a 2D POLYLINE entity with start width, end width and bulge value support.

Format codes:

x	x-coordinate
y	y-coordinate
s	start width
e	end width
b	bulge value
v	(x, y) tuple (z-axis is ignored)

Parameters

- **points** – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuple, value order according to the *format* string, unset values default to `0`
- **format** – format: format string, default is `'xy'`
- **closed** – `True` creates a closed polyline
- **start_width** – default start width, default is `0`
- **end_width** – default end width, default is `0`
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_polyline(*vertices: Iterable[Sequence[float]]*, *closed: bool = False*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a 3D POLYLINE entity.

Parameters

- **vertices** – iterable of `(x, y[, z])` tuples, z-axis is `0` by default
- **closed** – `True` creates a closed polyline
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_polyface(*vertices: Iterable[Sequence[float]]*, *faces: Iterable[Sequence[int]]*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a POLYFACE entity. The POLYFACE entity supports only faces of maximum 4 vertices, more indices will be ignored. A simple square would be:

```
v0 = (0, 0, 0)
v1 = (1, 0, 0)
v2 = (1, 1, 0)
v3 = (0, 1, 0)
dxf.add_polyface(vertices=[v0, v1, v2, v3], faces=[(0, 1, 2, 3)])
```

All 3D form functions of the *ezdxf.render.forms* module return *MeshBuilder* objects, which provide the required vertex and face lists.

See sphere example: https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py

Parameters

- **vertices** – iterable of `(x, y, z)` tuples
- **faces** – iterable of 3 or 4 vertex indices, indices have to be 0-based
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_polymesh(*vertices: Iterable[Sequence[float]]*, *size: tuple[int, int]*, *closed=(False, False)*, *layer: str = '0'*, *color: int | None = None*, *linetype: str | None = None*) → None

Add a POLYMESH entity. A POLYMESH is a mesh of m rows and n columns, each mesh vertex has its own x-, y- and z coordinates. The mesh can be closed in m- and/or n-direction. The vertices have to be in column order: (m0, n0), (m0, n1), (m0, n2), (m1, n0), (m1, n1), (m1, n2), …

See example: https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py

Parameters

- **vertices** – iterable of `(x, y, z)` tuples, in column order
- **size** – mesh dimension as (m, n)-tuple, requirement: `len(vertices) == m*n`
- **closed** – (m_closed, n_closed) tuple, for closed mesh in m and/or n direction
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*
- **linetype** – line type as string see *add_line()*

add_text(*text: str*, *insert: Sequence[float] = (0, 0)*, *height: float = 1.0*, *width: float = 1.0*, *align: str = 'LEFT'*, *rotation: float = 0.0*, *oblique: float = 0.0*, *style: str = 'STANDARD'*, *layer: str = '0'*, *color: int | None = None*) → None

Add a one line TEXT entity.

Parameters

- **text** – the text as string
- **insert** – insert location as `(x, y)` tuple
- **height** – text height in drawing units
- **width** – text width as factor
- **align** – text alignment, see table below
- **rotation** – text rotation in degrees as float
- **oblique** – oblique in degrees as float, vertical = `0` (default)

- **style** – text style name as string, if FIXED-TABLES are written some predefined text styles are available, else text style is always 'STANDARD'.
- **layer** – layer name as string see *add_line()*
- **color** – color as *AutoCAD Color Index (ACI)* see *add_line()*

Vert/Horiz	Left	Center	Right
Top	TOP_LEFT	TOP_CENTER	TOP_RIGHT
Middle	MIDDLE_LEFT	MIDDLE_CENTER	MIDDLE_RIGHT
Bottom	BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT
Baseline	LEFT	CENTER	RIGHT

The special alignments ALIGNED and FIT are not available.

6.12.9 text2path

Tools to convert text strings and text based DXF entities into outer- and inner linear paths as *Path* objects. These tools depend on the optional Matplotlib package. At the moment only the TEXT and the ATTRIB entity can be converted into paths and hatches.

Don't expect a 100% match compared to CAD applications.

Text Alignments

The text alignments are enums of type *ezdxf.enums.TextEntityAlignment*

Vertical	Left	Center	Right
Top	TOP_LEFT	TOP_CENTER	TOP_RIGHT
Middle	MIDDLE_LEFT	MIDDLE_CENTER	MIDDLE_RIGHT
Bottom	BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT
Baseline	LEFT	CENTER	RIGHT

The vertical middle alignments (MIDDLE_XXX), center the text vertically in the middle of the uppercase letter "X" (cap height).

Special alignments, where the horizontal alignment is always in the center of the text:

- ALIGNED: text is scaled to match the given *length*, scales x- and y-direction by the same factor.
- FIT: text is scaled to match the given *length*, but scales only in x-direction.
- MIDDLE: insertion point is the center of the total height (cap height + descender height) without scaling, the *length* argument is ignored.

Font Face Definition

A font face is defined by the Matplotlib compatible *FontFace* object by `font-family`, `font-style`, `font-stretch` and `font-weight`.

See also:

- *Font Anatomy*
- *Font Properties*

String Functions

ezdxf.addons.text2path.**make_path_from_str**(*s: str*, *font:* FontFace, *size: float = 1.0*, *align=TextEntityAlignment.LEFT*, *length: float = 0*, *m:* Matrix44 = *None*) → Path

Convert a single line string *s* into a *Multi-Path* object. The text *size* is the height of the uppercase letter "X" (cap height). The paths are aligned about the insertion point at (0, 0). BASELINE means the bottom of the letter "X".

Parameters

- **s** – text to convert
- **font** – font face definition as *FontFace* object
- **size** – text size (cap height) in drawing units
- **align** – alignment as *ezdxf.enums.TextEntityAlignment*, default is `LEFT`
- **length** – target length for the `ALIGNED` and `FIT` alignments
- **m** – transformation *Matrix44*

ezdxf.addons.text2path.**make_paths_from_str**(*s: str*, *font:* FontFace, *size: float = 1.0*, *align=TextEntityAlignment.LEFT*, *length: float = 0*, *m:* Matrix44 = *None*) → list[ezdxf.path.path.Path]

Convert a single line string *s* into a list of *Path* objects. All paths are returned as a list of *Single-Path* objects. The text *size* is the height of the uppercase letter "X" (cap height). The paths are aligned about the insertion point at (0, 0). BASELINE means the bottom of the letter "X".

Parameters

- **s** – text to convert
- **font** – font face definition as *FontFace* object
- **size** – text size (cap height) in drawing units
- **align** – alignment as *ezdxf.enums.TextEntityAlignment*, default is `LEFT`
- **length** – target length for the `ALIGNED` and `FIT` alignments
- **m** – transformation *Matrix44*

ezdxf.addons.text2path.**make_hatches_from_str**(*s: str*, *font:* FontFace, *size: float = 1.0*, *align=TextEntityAlignment.LEFT*, *length: float = 0*, *dxfattribs=None*, *m:* Matrix44 = *None*) → list[ezdxf.entities.hatch.Hatch]

Convert a single line string *s* into a list of virtual *Hatch* entities. The text *size* is the height of the uppercase letter "X" (cap height). The paths are aligned about the insertion point at (0, 0). The HATCH entities are aligned to this insertion point. BASELINE means the bottom of the letter "X".

Parameters

- **s** – text to convert
- **font** – font face definition as *FontFace* object
- **size** – text size (cap height) in drawing units
- **align** – alignment as *ezdxf.enums.TextEntityAlignment*, default is LEFT
- **length** – target length for the ALIGNED and FIT alignments
- **dxfattribs** – additional DXF attributes
- **m** – transformation *Matrix44*

Entity Functions

class ezdxf.addons.text2path.**Kind**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The *Kind* enum defines the DXF types to create as bit flags, e.g. 1+2 to get HATCHES as filling and SPLINES and POLYLINES as outline:

Int	Enum	Description
1	HATCHES	*Hatch* entities as filling
2	SPLINES	*Spline* and 3D *Polyline* entities as outline
4	LWPOLYLINES	*LWPolyline* entities as approximated (flattened) outline

ezdxf.addons.text2path.**virtual_entities**(*entity: Text | Attrib*, *kind: int = Kind.HATCHES*) → *EntityQuery*

Convert the text content of DXF entities TEXT and ATTRIB into virtual SPLINE and 3D POLYLINE entities or approximated LWPOLYLINE entities as outlines, or as HATCH entities as fillings.

Returns the virtual DXF entities as an *EntityQuery* object.

Parameters

- **entity** – TEXT or ATTRIB entity
- **kind** – kind of entities to create as bit flags, see enum *Kind*

ezdxf.addons.text2path.**explode**(*entity: Text | Attrib*, *kind: int = Kind.HATCHES*, *target=None*) → *EntityQuery*

Explode the text *entity* into virtual entities, see *virtual_entities()*. The source entity will be destroyed.

The target layout is given by the *target* argument, if *target* is None, the target layout is the source layout of the text entity.

Returns the created DXF entities as an *EntityQuery* object.

Parameters

- **entity** – TEXT or ATTRIB entity to explode
- **kind** – kind of entities to create as bit flags, see enum *Kind*
- **target** – target layout for new created DXF entities, None for the same layout as the source entity.

ezdxf.addons.text2path.**make_path_from_entity**(*entity: Text | Attrib*) → Path

Convert text content from DXF entities TEXT and ATTRIB into a *Multi-Path* object. The paths are located at the location of the source entity.

ezdxf.addons.text2path.**make_paths_from_entity**(*entity: Text | Attrib*) → list[ezdxf.path.path.Path]

Convert text content from DXF entities TEXT and ATTRIB into a list of `Path` objects. All paths are returned as a list of *Single-Path* objects. The paths are located at the location of the source entity.

6.12.10 MTextExplode

This tool is meant to explode MTEXT entities into single line TEXT entities by replicating the MTEXT layout as close as possible. This tool requires the optional Matplotlib package to create usable results, nonetheless it also works without Matplotlib, but then uses a mono-spaced replacement font for text size measuring which leads to very inaccurate results.

The supported MTEXT features are:

- changing text color
- text strokes: underline, overline and strike through
- changing text size, width and oblique
- changing font faces
- stacked text (fractions)
- multi-column support
- background color
- text frame

The tool requires an initialized DXF document io implement all these features by creating additional text styles. When exploding multiple MTEXT entities, they can share this new text styles. Call the `MTextExplode.finalize()` method just once after all MTEXT entities are processed to create the required text styles, or use `MTextExplode` as context manager by using the `with` statement, see examples below.

There are also many limitations:

- A 100% accurate result cannot be achieved.
- Character tracking is not supported.
- Tabulator stops have only limited support for LEFT and JUSTIFIED aligned paragraphs to support numbered and bullet lists. An excessive use of tabs will lead to incorrect results.
- The DISTRIBUTED alignment will be replaced by the JUSTIFIED alignment.
- Text flow is always "left to right".
- The line spacing mostly corresponds to the "EXACT" style, except for stacked text (fractions), which corresponds more to the "AT LEAST" style, but not precisely. This behavior maybe will improve in the future.
- FIELDS are not evaluated by *ezdxf*.

class ezdxf.addons.**MTextExplode**(*layout*, *doc=None*, *spacing_factor=1.0*)

The `MTextExplode` class is a tool to disassemble MTEXT entities into single line TEXT entities and additional LINE entities if required to emulate strokes.

The *layout* argument defines the target layout for "exploded" parts of the MTEXT entity. Use argument *doc* if the target layout has no DXF document assigned like virtual layouts. The *spacing_factor* argument is an advanced tuning parameter to scale the size of space chars.

explode(*mtext: MText*, *destroy=True*)

Explode *mtext* and destroy the source entity if argument *destroy* is `True`.

finalize()

Create required text styles. This method is called automatically if the class is used as context manager. This method does not work with virtual layouts if no document was assigned at initialization!

Example to explode all MTEXT entities in the DXF file "mtext.dxf":

```
import ezdxf
from ezdxf.addons import MTextExplode

doc = ezdxf.readfile("mtext.dxf")
msp = doc.modelspace()
with MTextExplode(msp) as xpl:
    for mtext in msp.query("MTEXT"):
        xpl.explode(mtext)
doc.saveas("xpl_mtext.dxf")
```

Explode all MTEXT entities into the block "EXPLODE":

```
import ezdxf
from ezdxf.addons import MTextExplode

doc = ezdxf.readfile("mtext.dxf")
msp = doc.modelspace()
blk = doc.blocks.new("EXPLODE")
with MTextExplode(blk) as xpl:
    for mtext in msp.query("MTEXT"):
        xpl.explode(mtext)
msp.add_block_ref("EXPLODE", (0, 0))
doc.saveas("xpl_into_block.dxf")
```

6.12.11 PyCSG

Constructive Solid Geometry (CSG) is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. This library implements CSG operations on meshes elegantly and concisely using BSP trees, and is meant to serve as an easily understandable implementation of the algorithm. All edge cases involving overlapping coplanar polygons in both solids are correctly handled.

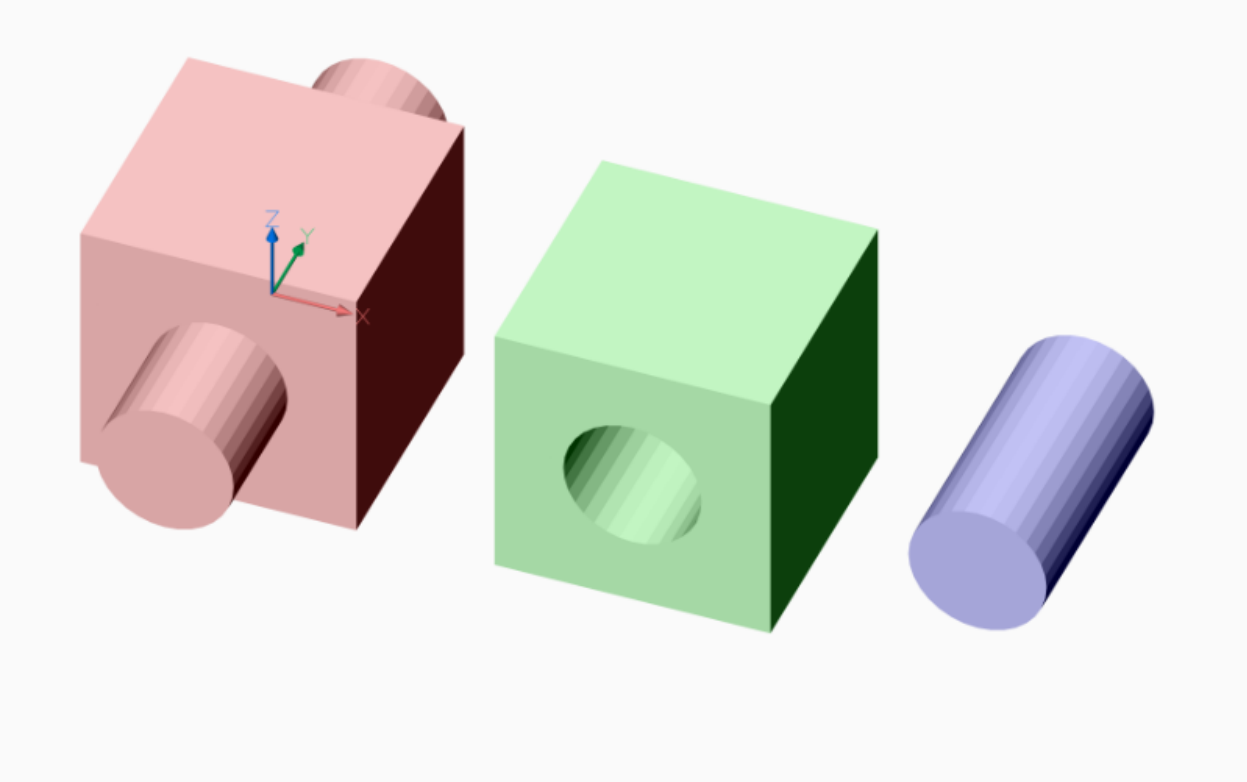
Example for usage:

```
import ezdxf
from ezdxf.render.forms import cube, cylinder_2p
from ezdxf.addons.pycsg import CSG

# create new DXF document
doc = ezdxf.new()
msp = doc.modelspace()

# create same geometric primitives as MeshTransformer() objects
cube1 = cube()
cylinder1 = cylinder_2p(count=32, base_center=(0, -1, 0), top_center=(0, 1, 0),␣
↪radius=.25)

# build solid union
```

(continues on next page)

(continued from previous page)

```
union = CSG(cube1) + CSG(cylinder1)
# convert to mesh and render mesh to modelspace
union.mesh().render(msp, dxfattribs={'color': 1})

# build solid difference
difference = CSG(cube1) - CSG(cylinder1)
# convert to mesh, translate mesh and render mesh to modelspace
difference.mesh().translate(1.5).render(msp, dxfattribs={'color': 3})

# build solid intersection
intersection = CSG(cube1) * CSG(cylinder1)
# convert to mesh, translate mesh and render mesh to modelspace
intersection.mesh().translate(2.75).render(msp, dxfattribs={'color': 5})

doc.saveas('csg.dxf')
```



This CSG kernel supports only meshes as *MeshBuilder* objects, which can be created from and converted to DXF *Mesh* entities.

This CSG kernel is **not** compatible with ACIS objects like *Solid3d*, *Body*, *Surface* or *Region*.

Note: This is a pure Python implementation, don't expect great performance and the implementation is based on an unbalanced BSP tree, so in the case of `RecursionError`, increase the recursion limit:

```
import sys

actual_limit = sys.getrecursionlimit()
# default is 1000, increasing too much may cause a seg fault
sys.setrecursionlimit(10000)
```

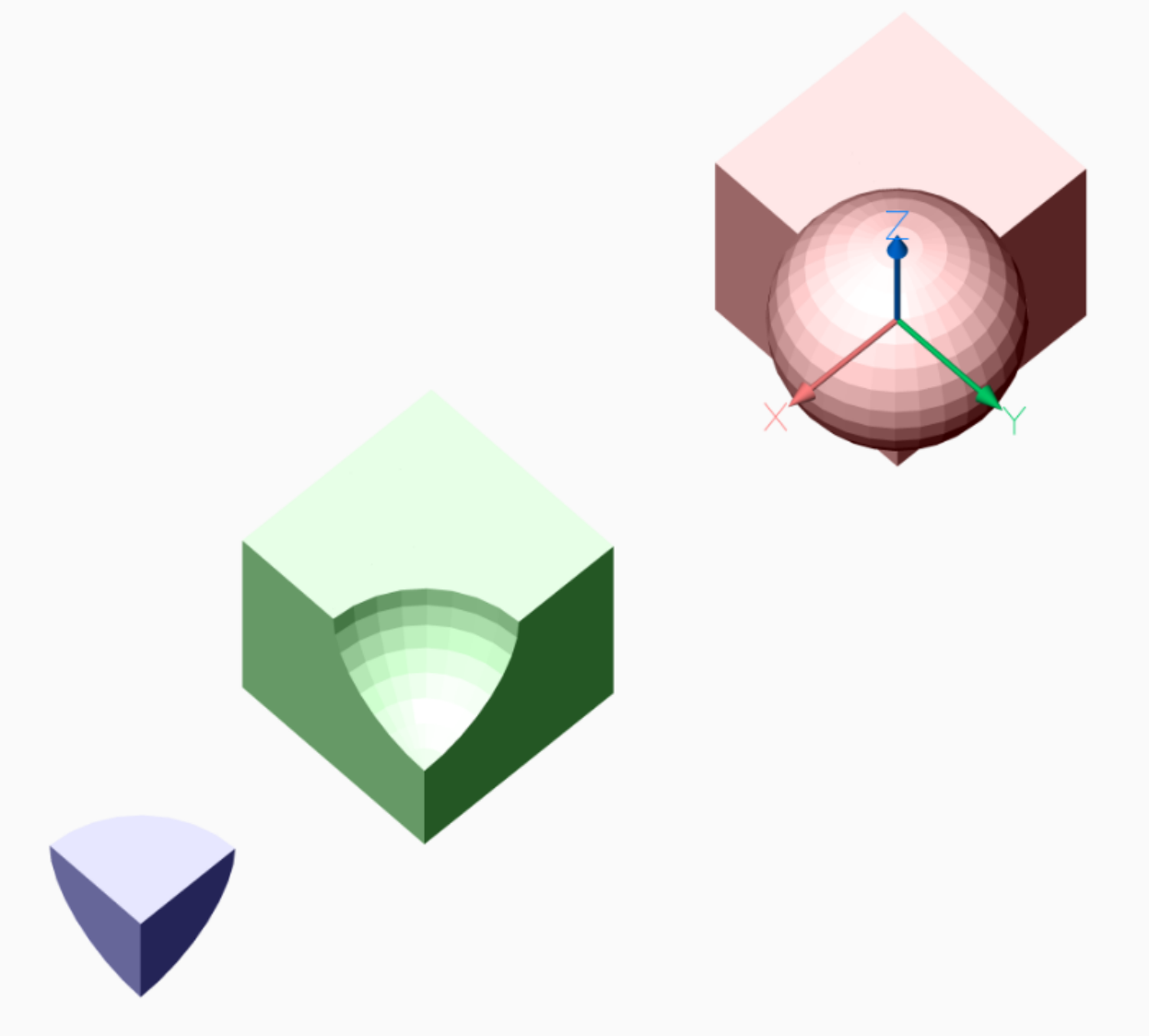
(continues on next page)

(continued from previous page)

```
...  # do the CSG stuff

sys.setrecursionlimit(actual_limit)
```

CSG works also with spheres, but with really bad runtime behavior and most likely `RecursionError` exceptions, and use quadrilaterals as body faces to reduce face count by setting argument *quads* to `True`.

```
import ezdxf

from ezdxf.render.forms import sphere, cube
from ezdxf.addons.pycsg import CSG

doc = ezdxf.new()
doc.set_modelspace_vport(6, center=(5, 0))
msp = doc.modelspace()

cube1 = cube().translate(-.5, -.5, -.5)
sphere1 = sphere(count=32, stacks=16, radius=.5, quads=True)

union = (CSG(cube1) + CSG(sphere1)).mesh()
union.render(msp, dxfattribs={'color': 1})

subtract = (CSG(cube1) - CSG(sphere1)).mesh().translate(2.5)
subtract.render(msp, dxfattribs={'color': 3})

intersection = (CSG(cube1) * CSG(sphere1)).mesh().translate(4)
intersection.render(msp, dxfattribs={'color': 5})
```



Hard Core CSG - Menger Sponge Level 3 vs Sphere

Required runtime on an old Xeon E5-1620 Workstation @ 3.60GHz, with default recursion limit of 1000 on Windows 10:

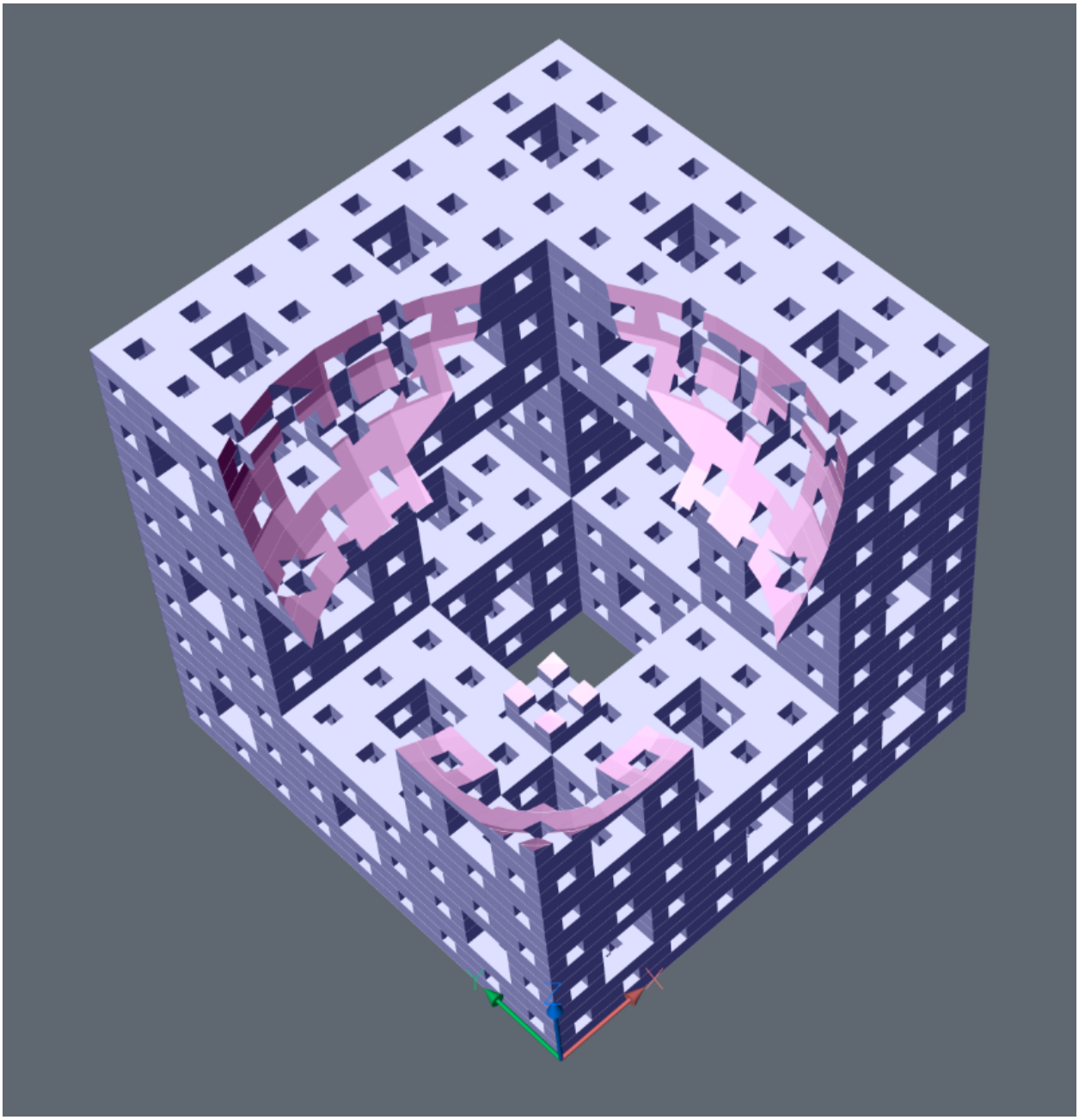
- CPython 3.8.1 64bit: ~60 seconds,
- pypy3 [PyPy 7.2.0] 32bit: ~6 seconds, and using `__slots__` reduced runtime below 5 seconds, yes - pypy is worth a look for long running scripts!

```
from ezdxf.render.forms import sphere
from ezdxf.addons import MengerSponge
from ezdxf.addons.pycsg import CSG

doc = ezdxf.new()
doc.layers.new('sponge', dxfattribs={'color': 5})
doc.layers.new('sphere', dxfattribs={'color': 6})

doc.set_modelspace_vport(6, center=(5, 0))
msp = doc.modelspace()
```

(continues on next page)

(continued from previous page)

```
sponge1 = MengerSponge(level=3).mesh()
sphere1 = sphere(count=32, stacks=16, radius=.5, quads=True).translate(.25, .25, 1)

subtract = (CSG(sponge1, meshid=1) - CSG(sphere1, meshid=2))
# get mesh result by id
subtract.mesh(1).render(msp, dxfattribs={'layer': 'sponge'})
subtract.mesh(2).render(msp, dxfattribs={'layer': 'sphere'})
```



CSG Class

class ezdxf.addons.pycsg.**CSG**(*mesh:* MeshBuilder, *meshid: int = 0*)

Constructive Solid Geometry (CSG) is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. This class implements CSG operations on meshes.

New 3D solids are created from *MeshBuilder* objects and results can be exported as *MeshTransformer* objects to *ezdxf* by method *mesh()*.

Parameters

- **mesh** – *ezdxf.render.MeshBuilder* or inherited object
- **meshid** – individual mesh ID to separate result meshes, 0 is default

mesh(*meshid: int = 0*) → MeshTransformer

Returns a *ezdxf.render.MeshTransformer* object.

Parameters
meshid – individual mesh ID, 0 is default

union(*other:* CSG) → *CSG*

Return a new CSG solid representing space in either this solid or in the solid *other*. Neither this solid nor the solid *other* are modified:

```
A.union(B)

+-------+             +-------+
|       |             |       |
|   A   |             |       |
|    +--+----+   =    |       +----+
+----+--+    |        +----+       |
     |   B   |             |       |
     |       |             |       |
     +-------+             +-------+
```

__add__()

```
union = A + B
```

subtract(*other:* CSG) → *CSG*

Return a new CSG solid representing space in this solid but not in the solid *other*. Neither this solid nor the solid *other* are modified:

```
A.subtract(B)

+-------+             +-------+
|       |             |       |
|   A   |             |       |
|    +--+----+   =    |    +--+
+----+--+    |        +----+
     |   B   |
     |       |
     +-------+
```

__sub__()

```
difference = A - B
```

intersect(*other:* CSG) → *CSG*

Return a new CSG solid representing space both this solid and in the solid *other*. Neither this solid nor the solid *other* are modified:

```
A.intersect(B)

+-------+
|       |
|   A   |
|    +--+----+   =   +--+
+----+--+    |       +--+
     |   B   |
     |       |
     +-------+
```

__mul__()

```
intersection = A * B
```

inverse() → *CSG*

Return a new CSG solid with solid and empty space switched. This solid is not modified.

License

- Original implementation csg.js, Copyright (c) 2011 Evan Wallace (http://madebyevan.com/), under the MIT license.
- Python port pycsg, Copyright (c) 2012 Tim Knip (http://www.floorplanner.com), under the MIT license.
- Additions by Alex Pletzer (Pennsylvania State University)
- Integration as *ezdxf* add-on, Copyright (c) 2020, Manfred Moitzi, MIT License.

6.12.12 Plot Style Files (CTB/STB)

CTB and STB files store plot styles used by AutoCAD and BricsCAD for printing and plotting.

If the plot style table is attached to a `Paperspace` or the `Modelspace`, a change of a plot style affects any object that uses that plot style. CTB files contain color dependent plot style tables, STB files contain named plot style tables.

See also:

- Using plot style tables in AutoCAD
- AutoCAD Plot Style Table Editor
- BricsCAD Plot Style Table Editor
- AUTODESK KNOWLEDGE NETWORK: How to install CTB files in AutoCAD

ezdxf.addons.acadctb.**load**(*filename: str*) → *ColorDependentPlotStyles* | *NamedPlotStyles*

Load the CTB or STB file *filename* from file system.

ezdxf.addons.acadctb.**new_ctb**() → *ColorDependentPlotStyles*

Create a new CTB file.

ezdxf.addons.acadctb.**new_stb**() → *NamedPlotStyles*

Create a new STB file.

ColorDependentPlotStyles

Color dependent plot style table (CTB file), table entries are `PlotStyle` objects.

class ezdxf.addons.acadctb.**ColorDependentPlotStyles**

description

Custom description of plot style file.

scale_factor

Specifies the factor by which to scale non-ISO linetypes and fill patterns.

apply_factor

Specifies whether or not you want to apply the `scale_factor`.

custom_lineweight_display_units

Set `1` for showing lineweight in inch in AutoCAD CTB editor window, but lineweights are always defined in millimeters.

lineweights

Lineweights table as `array.array`

__getitem__(*aci: int*) → *PlotStyle*

Returns `PlotStyle` for *AutoCAD Color Index (ACI)* *aci*.

__iter__()

Iterable of all plot styles.

new_style(*aci: int*, *data: dict | None = None*) → *PlotStyle*

Set *aci* to new attributes defined by *data* dict.

Parameters

- **aci** – *AutoCAD Color Index (ACI)*
- **data** – `dict` of `PlotStyle` attributes: description, color, physical_pen_number, virtual_pen_number, screen, linepattern_size, linetype, adaptive_linetype, lineweight, end_style, join_style, fill_style

get_lineweight(*aci: int*)

Returns the assigned lineweight for `PlotStyle` *aci* in millimeter.

get_lineweight_index(*lineweight: float*) → int

Get index of *lineweight* in the lineweight table or append *lineweight* to lineweight table.

get_table_lineweight(*index: int*) → float

Returns lineweight in millimeters of lineweight table entry *index*.

Parameters
index – lineweight table index = `PlotStyle.lineweight`

Returns
lineweight in mm or `0.0` for use entity lineweight

set_table_lineweight(*index: int*, *lineweight: float*) → int

Argument *index* is the lineweight table index, not the *AutoCAD Color Index (ACI)*.

Parameters

- **index** – lineweight table index = *PlotStyle.lineweight*
- **lineweight** – in millimeters

save()

Save CTB file as *filename* to the file system.

write(*stream: BinaryIO*) → None

Compress and write CTB file to binary *stream*.

NamedPlotStyles

Named plot style table (STB file), table entries are *PlotStyle* objects.

class ezdxf.addons.acadctb.**NamedPlotStyles**

description

Custom description of plot style file.

scale_factor

Specifies the factor by which to scale non-ISO linetypes and fill patterns.

apply_factor

Specifies whether or not you want to apply the *scale_factor*.

custom_lineweight_display_units

Set `1` for showing lineweight in inch in AutoCAD CTB editor window, but lineweights are always defined in millimeters.

lineweights

Lineweights table as `array.array`

__getitem__(*name: str*) → *PlotStyle*

Returns *PlotStyle* by *name*.

__delitem__(*name: str*) → None

Delete plot style *name*. Plot style `'Normal'` is not deletable.

__iter__() → Iterable[str]

Iterable of all plot style names.

new_style(*name: str*, *data: dict | None = None*, *localized_name: str | None = None*) → *PlotStyle*

Create new class:*PlotStyle name* by attribute dict *data*, replaces existing class:*PlotStyle* objects.

Parameters

- **name** – plot style name
- **localized_name** – name shown in plot style editor, uses *name* if `None`
- **data** – `dict` of *PlotStyle* attributes: description, color, physical_pen_number, virtual_pen_number, screen, linepattern_size, linetype, adaptive_linetype, lineweight, end_style, join_style, fill_style

get_lineweight(*name: str*)

Returns the assigned lineweight for `PlotStyle` *name* in millimeter.

get_lineweight_index(*lineweight: float*) → int

Get index of *lineweight* in the lineweight table or append *lineweight* to lineweight table.

get_table_lineweight(*index: int*) → float

Returns lineweight in millimeters of lineweight table entry *index*.

Parameters
index – lineweight table index = `PlotStyle.lineweight`

Returns
lineweight in mm or `0.0` for use entity lineweight

set_table_lineweight(*index: int*, *lineweight: float*) → int

Argument *index* is the lineweight table index, not the *AutoCAD Color Index (ACI)*.

Parameters

- **index** – lineweight table index = `PlotStyle.lineweight`
- **lineweight** – in millimeters

save()

Save STB file as *filename* to the file system.

write()

Compress and write STB file to binary *stream*.

PlotStyle

class ezdxf.addons.acadctb.**PlotStyle**

index

Table index (0-based). (int)

aci

AutoCAD Color Index (ACI) in range from `1` to `255`. Has no meaning for named plot styles. (int)

description

Custom description of plot style. (str)

physical_pen_number

Specifies physical plotter pen, valid range from `1` to `32` or `AUTOMATIC`. (int)

virtual_pen_number

Only used by non-pen plotters and only if they are configured for virtual pens. valid range from `1` to `255` or `AUTOMATIC`. (int)

screen

Specifies the color intensity of the plot on the paper, valid range is from `0` to `100`. (int)

If you select `100` the drawing will plotted with its full color intensity. In order for screening to work, the `dithering` option must be active.

linetype

Overrides the entity linetype, default value is `OBJECT_LINETYPE`. (bool)

adaptive_linetype

`True` if a complete linetype pattern is more important than a correct linetype scaling, default is `True`. (bool)

linepattern_size

Line pattern size, default = `0.5`. (float)

lineweight

Overrides the entity lineWEIGHT, default value is *OBJECT_LINEWEIGHT*. This is an index into the `UserStyles.lineweights` table. (int)

end_style

Line end cap style, see table below, default is `END_STYLE_OBJECT` (int)

join_style

Line join style, see table below, default is `JOIN_STYLE_OBJECT` (int)

fill_style

Line fill style, see table below, default is `FILL_STYLE_OBJECT` (int)

dithering

Depending on the capabilities of your plotter, dithering approximates the colors with dot patterns. When this option is `False`, the colors are mapped to the nearest color, resulting in a smaller range of colors when plotting.

Dithering is available only whether you select the object's color or assign a plot style color.

grayscale

Plot colors in grayscale. (bool)

Default Line Weights

#	[mm]
0	0.00
1	0.05
2	0.09
3	0.10
4	0.13
5	0.15
6	0.18
7	0.20
8	0.25
9	0.30
10	0.35
11	0.40
12	0.45
13	0.50
14	0.53
15	0.60
16	0.65
17	0.70
18	0.80
19	0.90
20	1.00
21	1.06
22	1.20
23	1.40
24	1.58
25	2.00
26	2.11

Predefined Values

ezdxf.addons.acadctb.**AUTOMATIC**

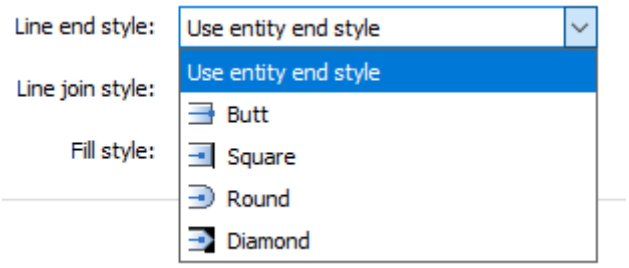
ezdxf.addons.acadctb.**OBJECT_LINEWEIGHT**

ezdxf.addons.acadctb.**OBJECT_LINETYPE**

ezdxf.addons.acadctb.**OBJECT_COLOR**

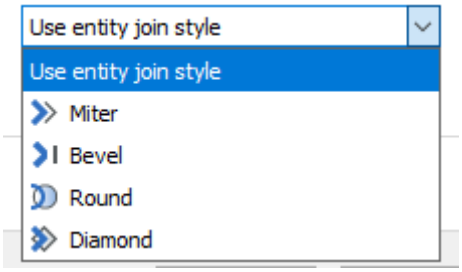
ezdxf.addons.acadctb.**OBJECT_COLOR2**

Line End Style



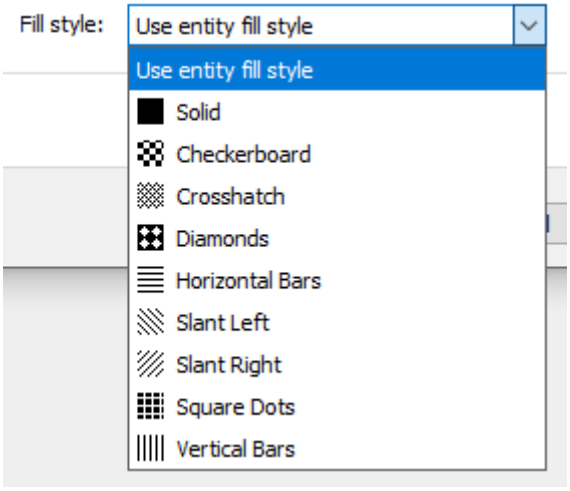
END_STYLE_BUTT	0
END_STYLE_SQUARE	1
END_STYLE_ROUND	2
END_STYLE_DIAMOND	3
END_STYLE_OBJECT	4

Line Join Style



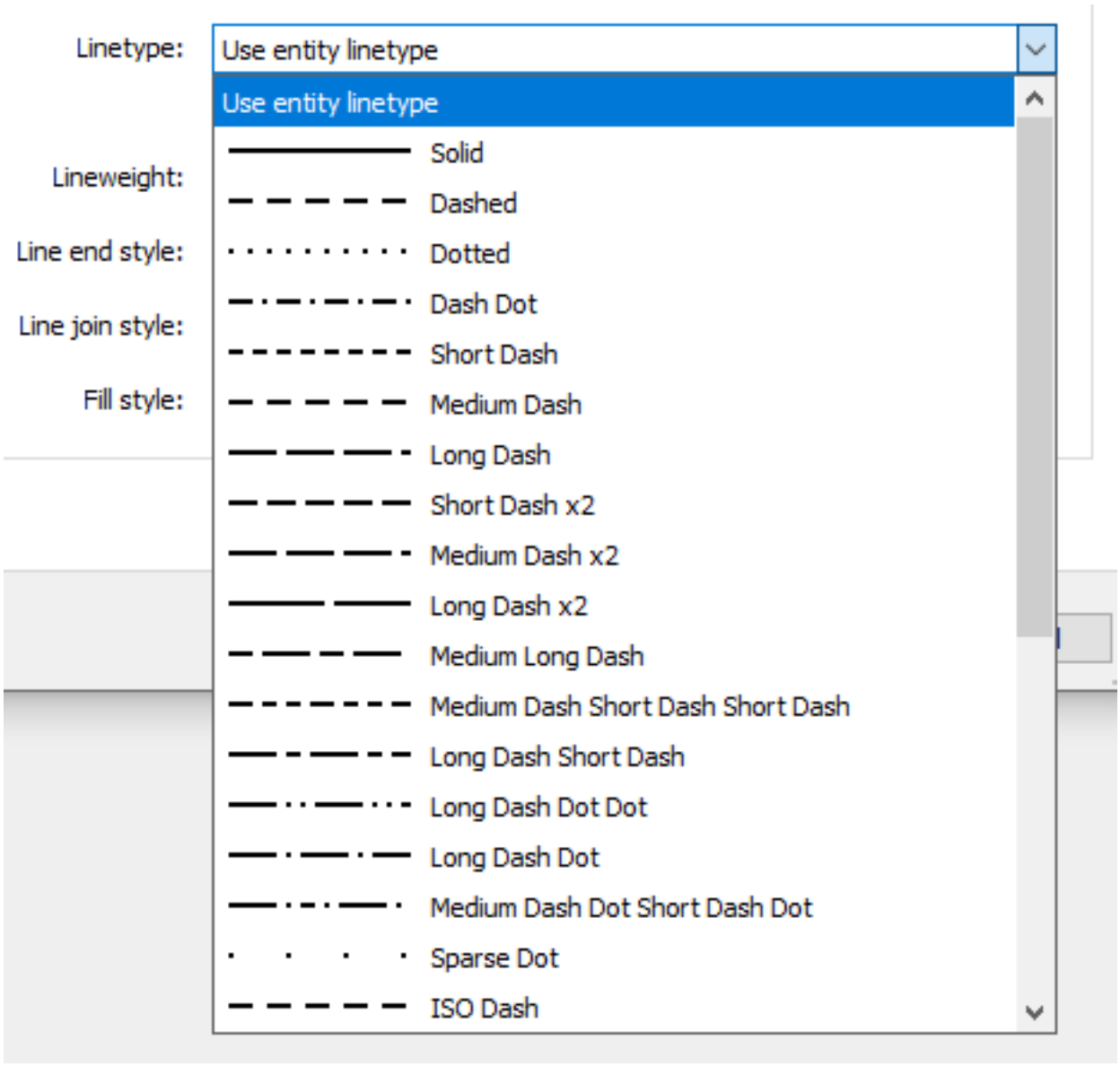
JOIN_STYLE_MITER	0
JOIN_STYLE_BEVEL	1
JOIN_STYLE_ROUND	2
JOIN_STYLE_DIAMOND	3
JOIN_STYLE_OBJECT	5

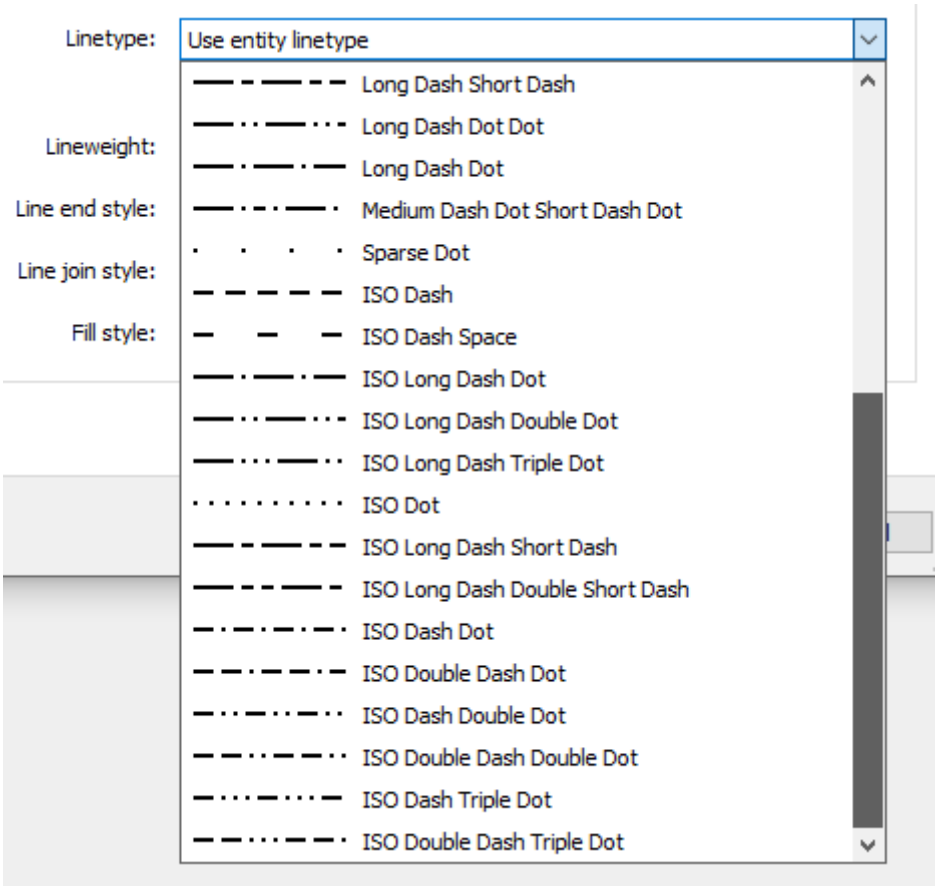
Fill Style



FILL_STYLE_SOLID	64
FILL_STYLE_CHECKERBOARD	65
FILL_STYLE_CROSSHATCH	66
FILL_STYLE_DIAMONDS	67
FILL_STYLE_HORIZONTAL_BARS	68
FILL_STYLE_SLANT_LEFT	69
FILL_STYLE_SLANT_RIGHT	70
FILL_STYLE_SQUARE_DOTS	71
FILL_STYLE_VERICAL_BARS	72
FILL_STYLE_OBJECT	73

Linetypes


Linetype:
Use entity linetype
Lineweight:
Line end style:
Line join style:
Fill style:
Use entity linetype
Solid
Dashed
Dotted
Dash Dot
Short Dash
Medium Dash
Long Dash
Short Dash x2
Medium Dash x2
Long Dash x2
Medium Long Dash
Medium Dash Short Dash Short Dash
Long Dash Short Dash
Long Dash Dot Dot
Long Dash Dot
Medium Dash Dot Short Dash Dot
Sparse Dot
ISO Dash



Linetype name	Value
Solid	0
Dashed	1
Dotted	2
Dash Dot	3
Short Dash	4
Medium Dash	5
Long Dash	6
Short Dash x2	7
Medium Dash x2	8
Long Dash x2	9
Medium Lang Dash	10
Medium Dash Short Dash Short Dash	11
Long Dash Short Dash	12
Long Dash Dot Dot	13
Long Dash Dot	14
Medium Dash Dot Short Dash Dot	15
Sparse Dot	16
ISO Dash	17
ISO Dash Space	18
ISO Long Dash Dot	19
ISO Long Dash Double Dot	20
ISO Long Dash Triple Dot	21

continues on next page

Table 3 – continued from previous page

Linetype name	Value
ISO Dot	22
ISO Long Dash Short Dash	23
ISO Long Dash Double Short Dash	24
ISO Dash Dot	25
ISO Double Dash Dot	26
ISO Dash Double Dot	27
ISO Double Dash Double Dot	28
ISO Dash Triple Dot	29
ISO Double Dash Triple Dot	30
Use entity linetype	31

6.12.13 Showcase Forms

MengerSponge

Build a 3D Menger sponge.

class ezdxf.addons.**MengerSponge**(*location: UVec = (0.0, 0.0, 0.0)*, *length: float = 1.0*, *level: int = 1*, *kind: int = 0*)

Parameters

- **location** – location of lower left corner as (x, y, z) tuple
- **length** – side length
- **level** – subdivide level
- **kind** – type of menger sponge

0	Original Menger Sponge
1	Variant XOX
2	Variant OXO
3	Jerusalem Cube

render(*layout: GenericLayoutType*, *merge: bool = False*, *dxfattribs=None*, *matrix: Matrix44 | None = None*, *ucs: UCS | None = None*) → None

Renders the menger sponge into layout, set *merge* to `True` for rendering the whole menger sponge into one MESH entity, set *merge* to `False` for rendering the individual cubes of the menger sponge as MESH entities.

Parameters

- **layout** – DXF target layout
- **merge** – `True` for one MESH entity, `False` for individual MESH entities per cube
- **dxfattribs** – DXF attributes for the MESH entities
- **matrix** – apply transformation matrix at rendering
- **ucs** – apply UCS transformation at rendering

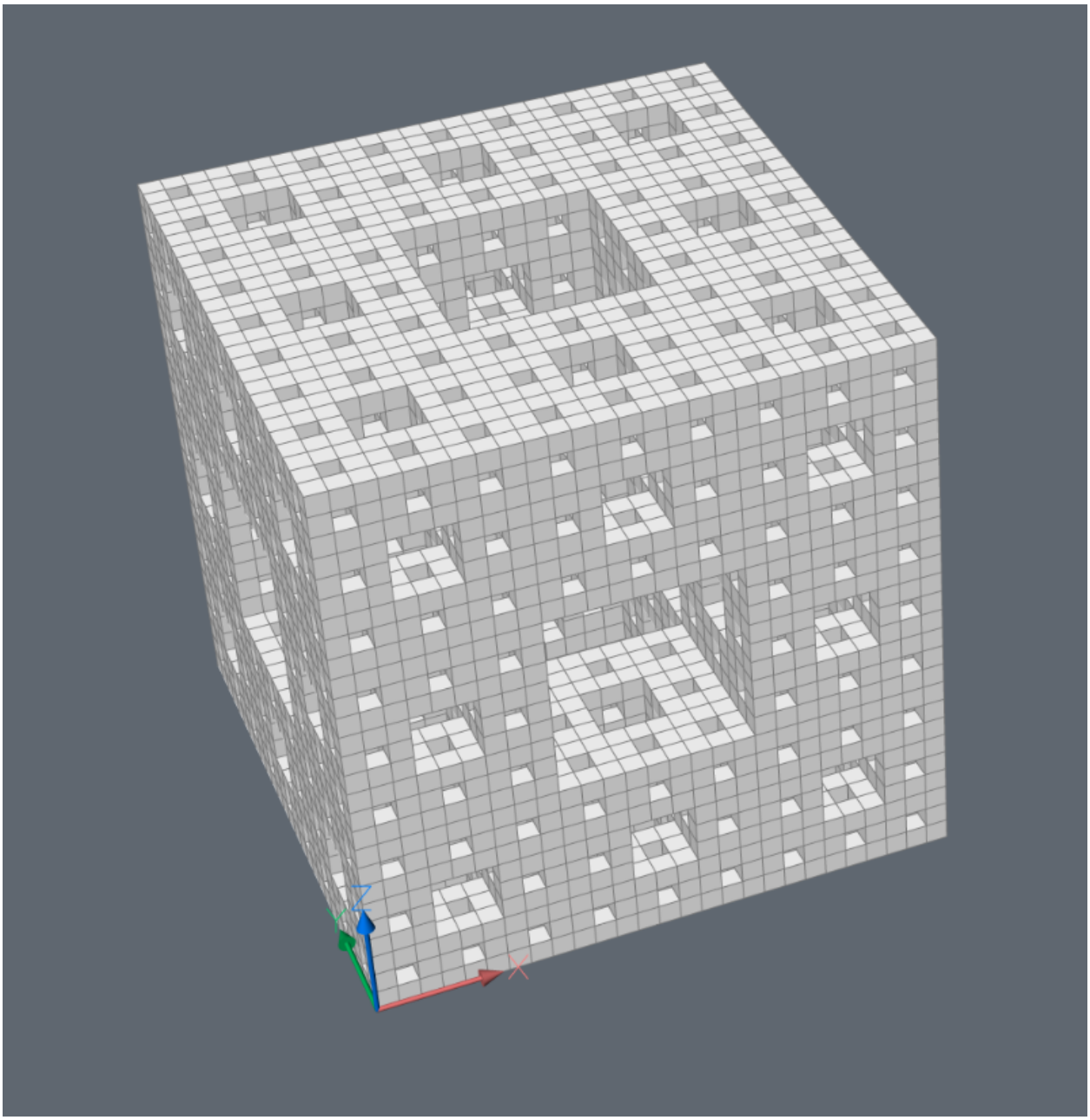
cubes() → Iterator[MeshTransformer]

Yields all cubes of the menger sponge as individual `MeshTransformer` objects.

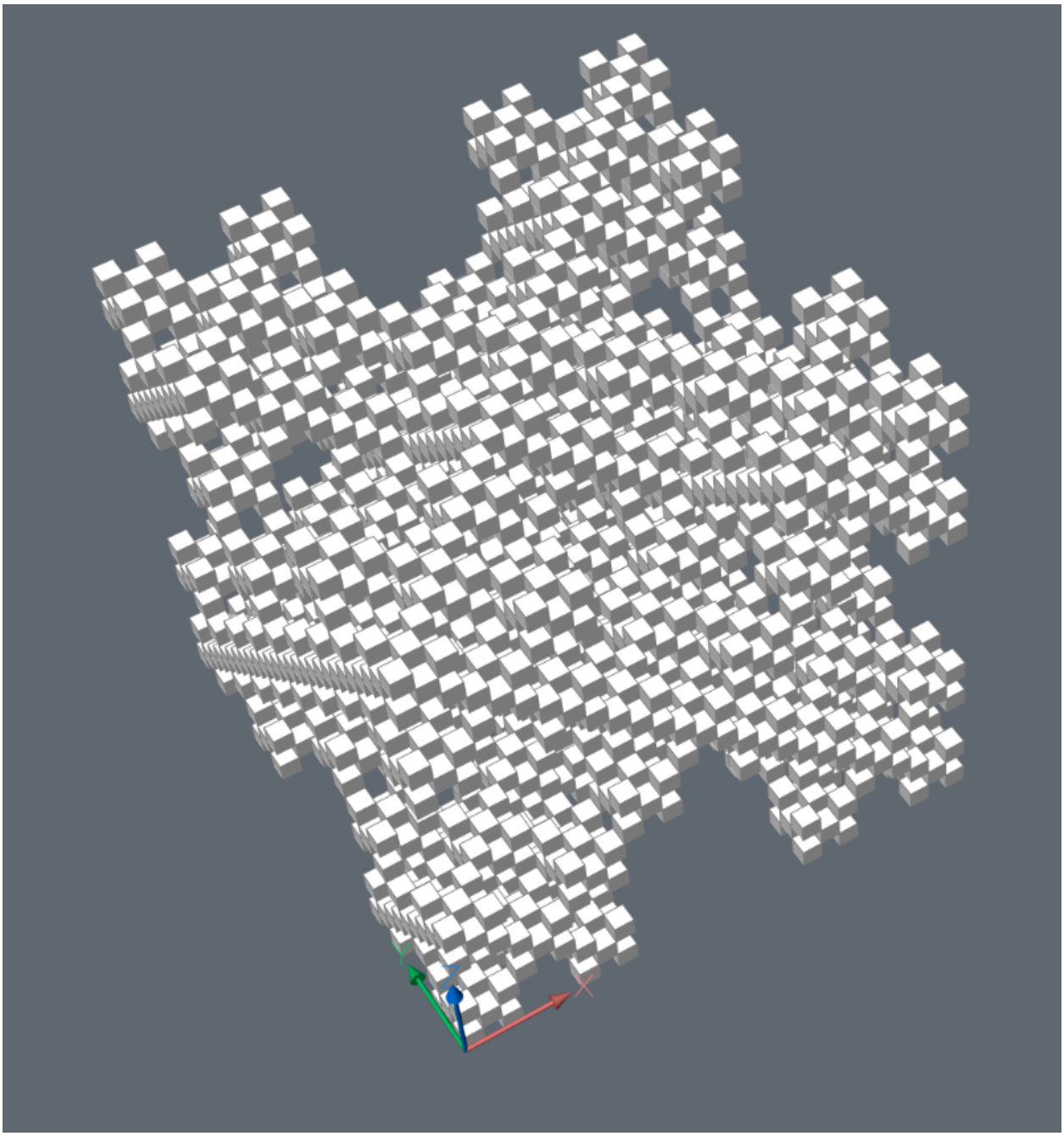
mesh() → MeshTransformer

Returns geometry as one `MeshTransformer` object.

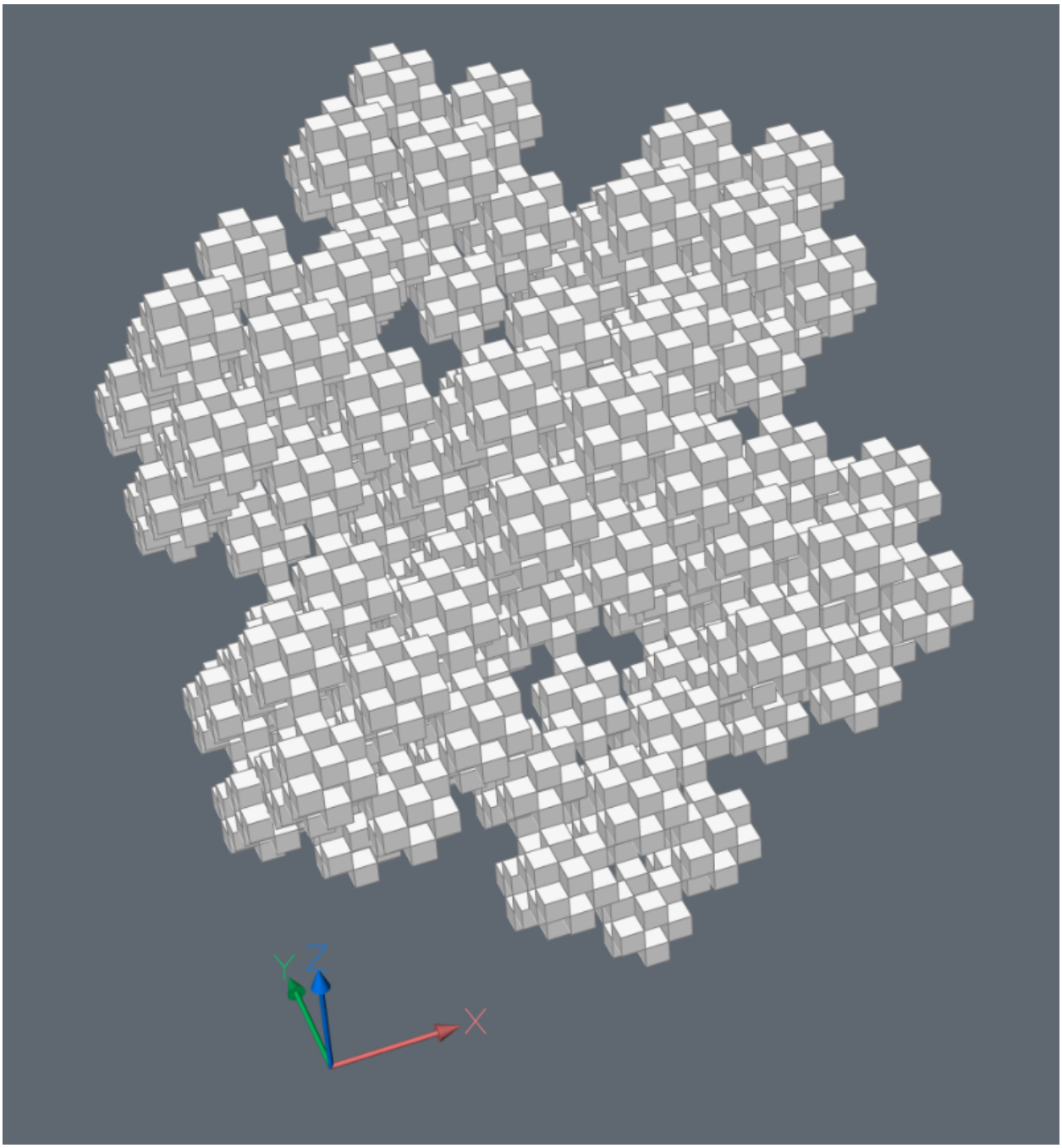
Menger Sponge `kind=0`:



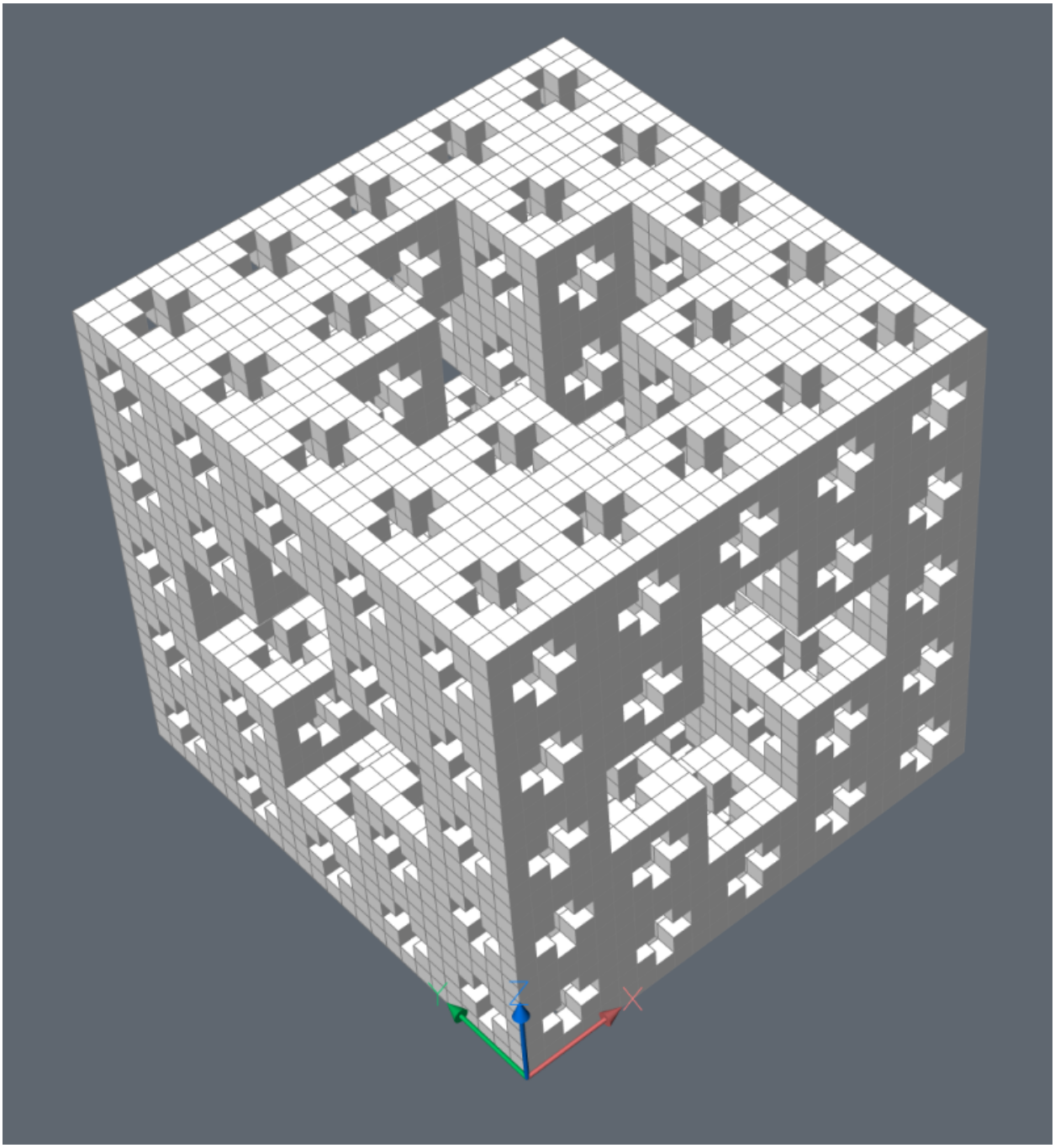
Menger Sponge `kind=1`:



Menger Sponge `kind=2`:



Jerusalem Cube `kind=3`:


Y
Z
X

SierpinskyPyramid

Build a 3D Sierpinsky Pyramid.

class ezdxf.addons.**SierpinskyPyramid**(*location:* UVec *= (0.0, 0.0, 0.0)*, *length: float = 1.0*, *level: int = 1*, *sides: int = 4*)

Parameters

- **location** – location of base center as (x, y, z) tuple
- **length** – side length
- **level** – subdivide level
- **sides** – sides of base geometry

render(*layout: GenericLayoutType*, *merge: bool = False*, *dxfattribs=None*, *matrix:* Matrix44 | *None = None*, *ucs:* UCS | *None = None*) → None

Renders the sierpinsky pyramid into layout, set *merge* to `True` for rendering the whole sierpinsky pyramid into one MESH entity, set *merge* to `False` for individual pyramids as MESH entities.

Parameters

- **layout** – DXF target layout
- **merge** – `True` for one MESH entity, `False` for individual MESH entities per pyramid
- **dxfattribs** – DXF attributes for the MESH entities
- **matrix** – apply transformation matrix at rendering
- **ucs** – apply UCS at rendering

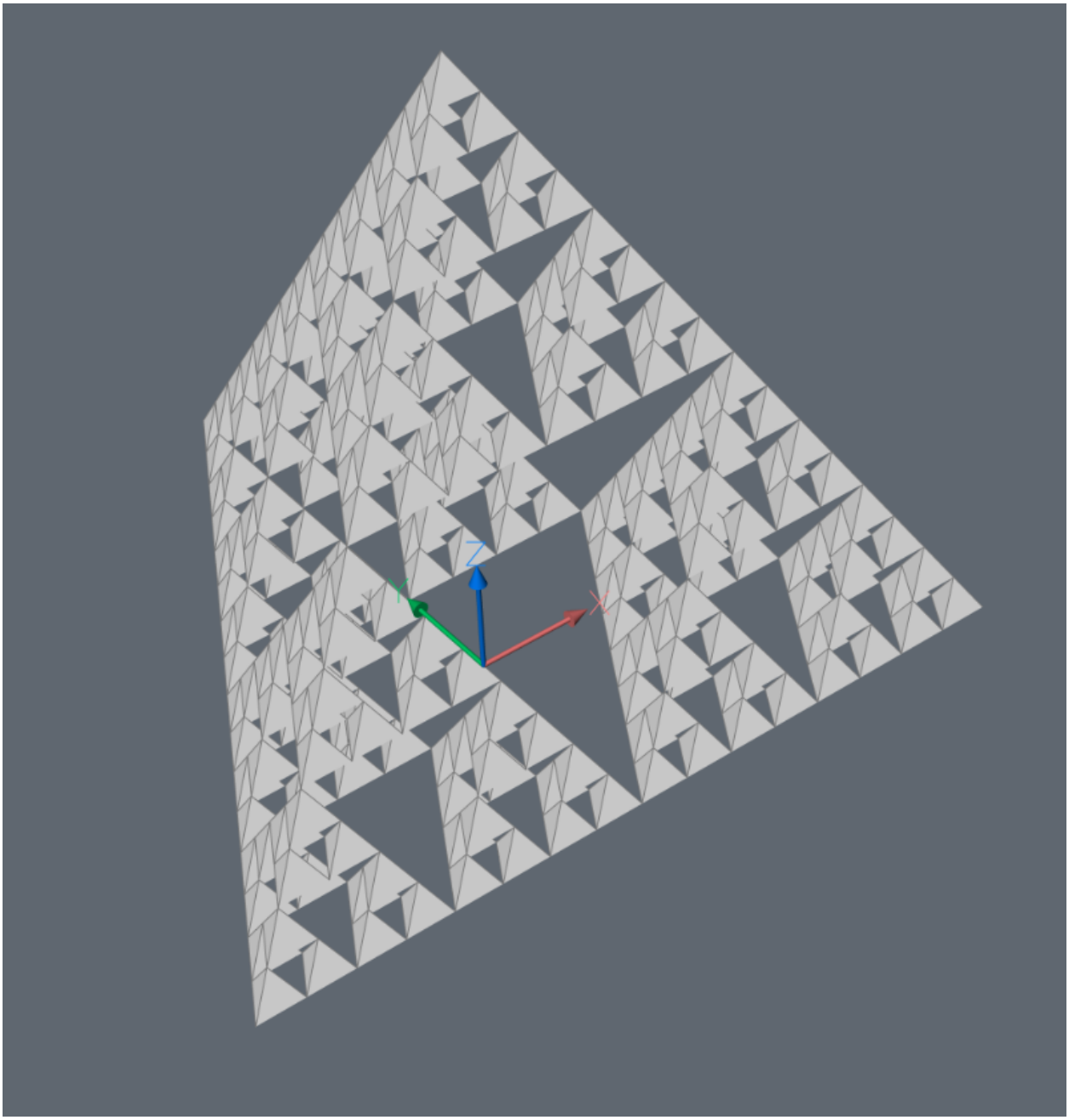
pyramids() → Iterable[MeshTransformer]

Yields all pyramids of the sierpinsky pyramid as individual `MeshTransformer` objects.

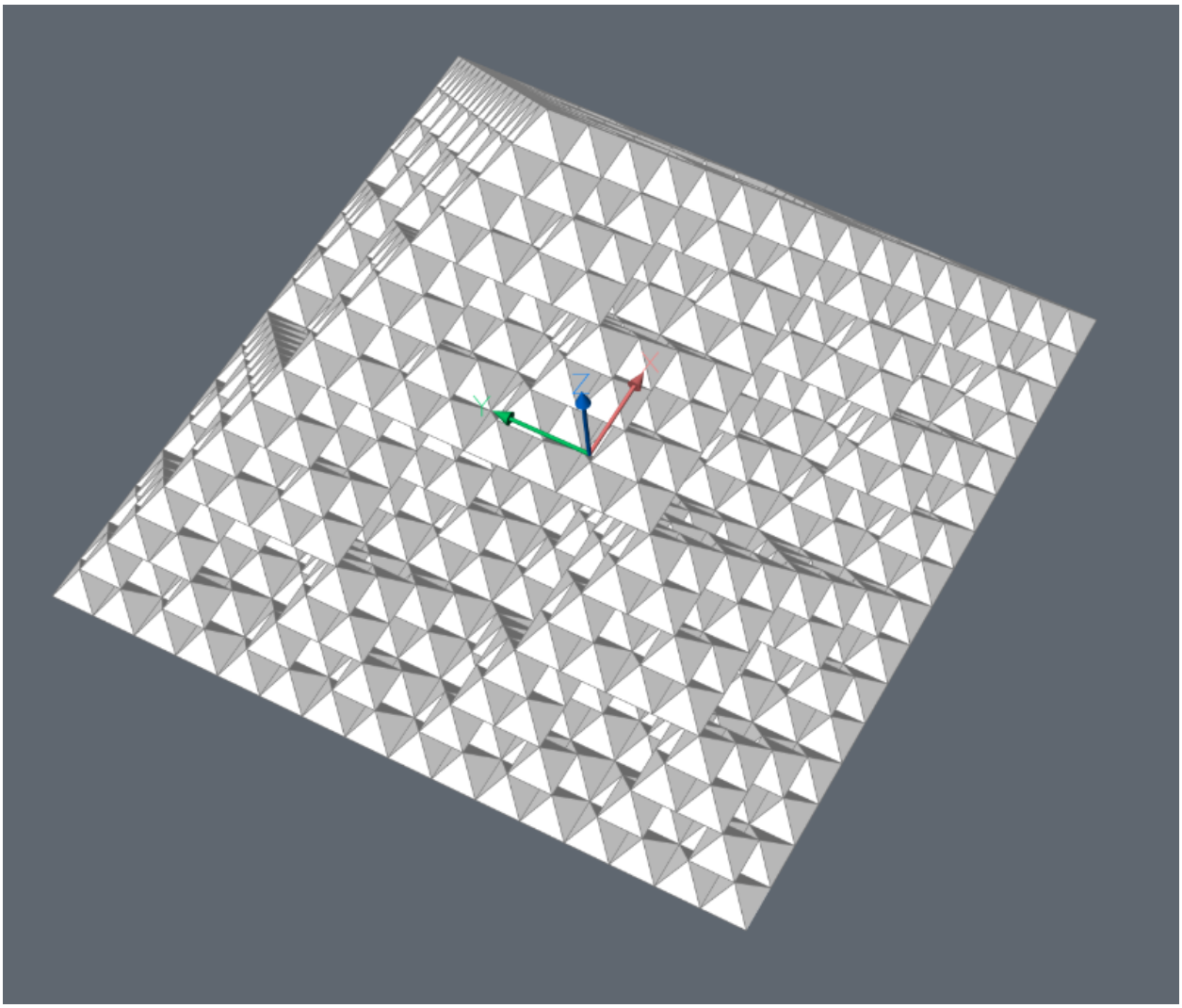
mesh() → MeshTransformer

Returns geometry as one `MeshTransformer` object.

Sierpinsky Pyramid with triangle base:



Sierpinsky Pyramid with square base:



6.12.14 Bin-Packing Add-on

This add-on is based on the 3D bin packing module py3dbp hosted on PyPI. Both sources of this package are MIT licensed like *ezdxf* itself.

The Bin Packing Problem

Quote from the Wikipedia article:

> The bin packing problem is an optimization problem, in which items of different sizes must be packed into a finite number of bins or containers, each of a fixed given capacity, in a way that minimizes the number of bins used.

Example

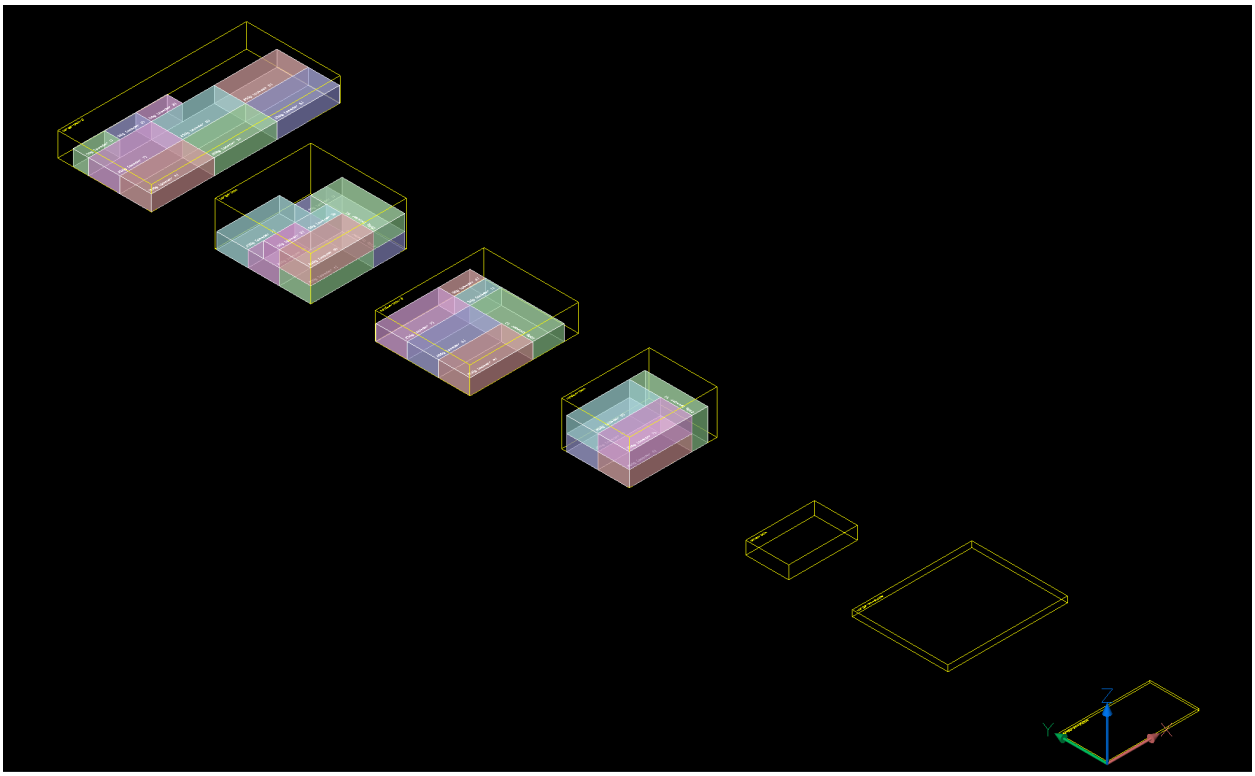
This code replicates the example used by the py3dbp package:

```
from typing import List
import ezdxf
from ezdxf import colors
from ezdxf.addons import binpacking as bp

SMALL_ENVELOPE = ("small-envelope", 11.5, 6.125, 0.25, 10)
LARGE_ENVELOPE = ("large-envelope", 15.0, 12.0, 0.75, 15)
SMALL_BOX = ("small-box", 8.625, 5.375, 1.625, 70.0)
MEDIUM_BOX = ("medium-box", 11.0, 8.5, 5.5, 70.0)
MEDIUM_BOX2 = ("medium-box-2", 13.625, 11.875, 3.375, 70.0)
LARGE_BOX = ("large-box", 12.0, 12.0, 5.5, 70.0)
LARGE_BOX2 = ("large-box-2", 23.6875, 11.75, 3.0, 70.0)

ALL_BINS = [
    SMALL_ENVELOPE,
    LARGE_ENVELOPE,
    SMALL_BOX,
    MEDIUM_BOX,
    MEDIUM_BOX2,
    LARGE_BOX,
    LARGE_BOX2,
]


def build_packer():
    packer = bp.Packer()
    packer.add_item("50g [powder 1]", 3.9370, 1.9685, 1.9685, 1)
    packer.add_item("50g [powder 2]", 3.9370, 1.9685, 1.9685, 2)
    packer.add_item("50g [powder 3]", 3.9370, 1.9685, 1.9685, 3)
    packer.add_item("250g [powder 4]", 7.8740, 3.9370, 1.9685, 4)
    packer.add_item("250g [powder 5]", 7.8740, 3.9370, 1.9685, 5)
    packer.add_item("250g [powder 6]", 7.8740, 3.9370, 1.9685, 6)
    packer.add_item("250g [powder 7]", 7.8740, 3.9370, 1.9685, 7)
    packer.add_item("250g [powder 8]", 7.8740, 3.9370, 1.9685, 8)
    packer.add_item("250g [powder 9]", 7.8740, 3.9370, 1.9685, 9)
    return packer


def make_doc():
    doc = ezdxf.new()
    doc.layers.add("FRAME", color=colors.YELLOW)
    doc.layers.add("ITEMS")
    doc.layers.add("TEXT")
    return doc


def main(filename):
    bins: List[bp.Bin] = []
    for box in ALL_BINS:
        packer = build_packer()
        packer.add_bin(*box)
        packer.pack(bp.PickStrategy.BIGGER_FIRST)
        bins.extend(packer.bins)
    doc = make_doc()
```

(continues on next page)

(continued from previous page)

```
    bp.export_dxf(doc.modelspace(), bins, offset=(0, 20, 0))
    doc.saveas(filename)


if __name__ == "__main__":
    main("py3dbp_example.dxf")
```



See also:

- example1 script
- example2 script

Packer Classes

class ezdxf.addons.binpacking.**AbstractPacker**

bins

List of containers to fill.

items

List of items to pack into the *bins*.

property is_packed: bool

Returns `True` if packer is packed, each packer can only be used once.

property unfitted_items: list[*ezdxf.addons.binpacking.Item*]

Returns the unfitted items.

__str__() → str
Return str(self).

append_bin(*box:* Bin) → None
Append a container.

append_item(*item:* Item) → None
Append a item.

get_fill_ratio() → float
Return the fill ratio of all bins.

get_capacity() → float
Returns the maximum fill volume of all bins.

get_total_weight() → float
Returns the total weight of all fitted items in all bins.

get_total_volume() → float
Returns the total volume of all fitted items in all bins.

pack(*pick=PickStrategy.BIGGER_FIRST*) → None
Pack items into bins. Distributes all items across all bins.

Packer

class ezdxf.addons.binpacking.**Packer**
3D Packer inherited from *AbstractPacker*.

add_bin(*name: str*, *width: float*, *height: float*, *depth: float*, *max_weight: float = UNLIMITED_WEIGHT*) → *Box*
Add a 3D *Box* container.

add_item(*payload*, *width: float*, *height: float*, *depth: float*, *weight: float = 0.0*) → *Item*
Add a 3D *Item* to pack.

FlatPacker

class ezdxf.addons.binpacking.**FlatPacker**
2D Packer inherited from *AbstractPacker*. All containers and items used by this packer must have a depth of 1.

add_bin(*name: str*, *width: float*, *height: float*, *max_weight: float = UNLIMITED_WEIGHT*) → *Envelope*
Add a 2D *Envelope* container.

add_item(*payload*, *width: float*, *height: float*, *weight: float = 0.0*) → *Item*
Add a 2D *FlatItem* to pack.

Bin Classes

class ezdxf.addons.binpacking.**Bin**(*name*, *width: float*, *height: float*, *depth: float*, *max_weight: float* = *UNLIMITED_WEIGHT*)

name
Name of then container as string.

width

height

depth

max_weight

property is_empty: bool

__str__() → str
Return str(self).

copy()
Returns a copy.

reset()
Reset the container to empty state.

put_item(*item:* Item, *pivot: tuple[float, float, float]*) → bool

get_capacity() → float
Returns the maximum fill volume of the bin.

get_total_weight() → float
Returns the total weight of all fitted items.

get_total_volume() → float
Returns the total volume of all fitted items.

get_fill_ratio() → float
Return the fill ratio.

Box Class

class ezdxf.addons.binpacking.**Box**(*name*, *width: float*, *height: float*, *depth: float*, *max_weight: float* = *UNLIMITED_WEIGHT*)

3D container inherited from *Bin*.

Envelope Class

class ezdxf.addons.binpacking.**Envelope**(*name*, *width: float*, *height: float*, *max_weight: float = UNLIMITED_WEIGHT*)

2D container inherited from *Bin*.

Item Class

class ezdxf.addons.binpacking.**Item**(*payload*, *width: float*, *height: float*, *depth: float*, *weight: float = 0.0*)

3D container item.

payload

Arbitrary Python object.

width

height

depth

weight

property bbox: AbstractBoundingBox

property rotation_type: *RotationType*

property position: tuple[float, float, float]

Returns the position of then lower left corner of the item in the container, the lower left corner is the origin (0, 0, 0).

copy()

Returns a copy, all copies have a reference to the same payload object.

__str__()

Return str(self).

get_volume() → float

Returns the volume of the item.

get_dimension() → tuple[float, float, float]

Returns the item dimension according the *rotation_type*.

get_transformation() → *Matrix44*

Returns the transformation matrix to transform the source entity located with the minimum extension corner of its bounding box in (0, 0, 0) to the final location including the required rotation.

FlatItem Class

class ezdxf.addons.binpacking.**FlatItem**(*payload*, *width: float*, *height: float*, *weight: float = 0.0*)

2D container item, inherited from *Item*. Has a default depth of 1.0.

Functions

ezdxf.addons.binpacking.**shuffle_pack**(*packer:* AbstractPacker, *attempts: int*) → *AbstractPacker*

Random shuffle packing. Returns a new packer with the best packing result, the input packer is unchanged.

Enums

RotationType

class ezdxf.addons.binpacking.**RotationType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Rotation type of an item:

- W = width
- H = height
- D = depth

WHD

HWD

HDW

DHW

DWH

WDH

PickStrategy

class ezdxf.addons.binpacking.**PickStrategy**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Order of how to pick items for placement.

BIGGER_FIRST

SMALLER_FIRST

SHUFFLE

Credits

- py3dbp package by Enzo Ruiz Pelaez
- bp3d by *gedex* - github repository on which py3dbp is based, written in Go
- Optimizing three-dimensional bin packing through simulation (PDF)

6.12.15 MeshExchange

The *ezdxf.addons.meshex* module provides functions to exchange meshes with other tools in the following file formats:

- STL: import/export, supports only triangles as faces
- OFF: import/export, supports ngons as faces and is more compact than STL
- OBJ: import/export, supports ngons as faces and can contain multiple meshes in one file
- PLY: export only, supports ngons as faces
- OpenSCAD: export as polyhedron, supports ngons as faces
- IFC4: export only, supports ngons as faces

The source or target object is always a *MeshBuilder* instance and therefore the supported features are also limited by this class. Only vertices and faces are exchanged, colors, textures and explicit face- and vertex normals are lost.

Note: This add-on is not a replacement for a proper file format interface for this data formats! It's just a simple way to exchange meshes with other tools like OpenSCAD or MeshLab.

Warning: The meshes created by the *ezdxf.addons.pycsg* add-on are usually not suitable for export because they often violate the vertex-to-vertex rule: A vertex of a face cannot lie on the edge of another face. This was one of the reasons to create this addon to get an interface to OpenSCAD.

Example for a simple STL to DXF converter:

```
import sys
import ezdxf
from ezdxf.addons import meshex

try:
    mesh = meshex.stl_readfile("your.stl")
except (meshex.ParsingError, IOError) as e:
    print(str(e))
    sys.exit(1)

doc = ezdxf.new()
mesh.render_mesh(doc.modelspace())
doc.saveas("your.dxf")
```

See also:

Example script meshex_export.py at github.

Import

ezdxf.addons.meshex.**stl_readfile**(*filename: str | PathLike*) → MeshTransformer

Read ascii or binary STL file content as *ezdxf.render.MeshTransformer* instance.

Raises
ParsingError – vertex parsing error or invalid/corrupt data

ezdxf.addons.meshex.**stl_loads**(*content: str*) → MeshTransformer

Load a mesh from an ascii STL content string as *ezdxf.render.MeshTransformer* instance.

Raises
ParsingError – vertex parsing error

ezdxf.addons.meshex.**stl_loadb**(*buffer: bytes*) → MeshTransformer

Load a mesh from a binary STL data *ezdxf.render.MeshTransformer* instance.

Raises
ParsingError – invalid/corrupt data or not a binary STL file

ezdxf.addons.meshex.**off_readfile**(*filename: str | PathLike*) → MeshTransformer

Read OFF file content as *ezdxf.render.MeshTransformer* instance.

Raises
ParsingError – vertex or face parsing error

ezdxf.addons.meshex.**off_loads**(*content: str*) → MeshTransformer

Load a mesh from a OFF content string as *ezdxf.render.MeshTransformer* instance.

Raises
ParsingError – vertex or face parsing error

ezdxf.addons.meshex.**obj_readfile**(*filename: str | PathLike*) → list[ezdxf.render.mesh.MeshTransformer]

Read OBJ file content as list of *ezdxf.render.MeshTransformer* instances.

Raises
ParsingError – vertex or face parsing error

ezdxf.addons.meshex.**obj_loads**(*content: str*) → list[ezdxf.render.mesh.MeshTransformer]

Load one or more meshes from an OBJ content string as list of *ezdxf.render.MeshTransformer* instances.

Raises
ParsingError – vertex parsing error

Export

ezdxf.addons.meshex.**stl_dumps**(*mesh: MeshBuilder*) → str

Returns the STL data as string for the given *mesh*. This function triangulates the meshes automatically because the STL format supports only triangles as faces.

This function does not check if the mesh obey the STL format rules:

- The direction of the face normal is outward.
- The face vertices are listed in counter-clockwise order when looking at the object from the outside (right-hand rule).
- Each triangle must share two vertices with each of its adjacent triangles.

- The object represented must be located in the all-positive octant (non-negative and nonzero).

ezdxf.addons.meshex.**stl_dumpb**(*mesh: MeshBuilder*) → bytes

Returns the STL binary data as bytes for the given *mesh*.

For more information see function: *stl_dumps()*

ezdxf.addons.meshex.**off_dumps**(*mesh: MeshBuilder*) → str

Returns the OFF data as string for the given *mesh*. The OFF format supports ngons as faces.

ezdxf.addons.meshex.**obj_dumps**(*mesh: MeshBuilder*) → str

Returns the OBJ data as string for the given *mesh*. The OBJ format supports ngons as faces.

ezdxf.addons.meshex.**ply_dumpb**(*mesh: MeshBuilder*) → bytes

Returns the PLY binary data as bytes for the given *mesh*. The PLY format supports ngons as faces.

ezdxf.addons.meshex.**scad_dumps**(*mesh: MeshBuilder*) → str

Returns the OpenSCAD polyhedron definition as string for the given *mesh*. OpenSCAD supports ngons as faces.

Important: OpenSCAD requires the face normals pointing inwards, the method *flip_normals()* of the *MeshBuilder* class can flip the normals inplace.

---

ezdxf.addons.meshex.**ifc4_dumps**(*mesh: MeshBuilder*, *entity_type=IfcEntityType.POLYGON_FACE_SET*, **, layer: str = 'MeshExport'*, *color: tuple[float, float, float] = (1.0, 1.0, 1.0)*) → str

Returns the IFC4 string for the given *mesh*. The caller is responsible for checking if the mesh is a closed or open surface (e.g. `mesh.diagnose().euler_characteristic == 2`) and using the appropriate entity type.

Parameters

- **mesh** – *MeshBuilder*
- **entity_type** – *IfcEntityType*
- **layer** – layer name as string
- **color** – entity color as RGB tuple, values in the range [0,1]

Warning: IFC4 is a very complex data format and this is a minimal effort exporter, so the exported data may not be importable by all CAD applications.

The exported IFC4 data can be imported by the following applications:

- BricsCAD
- FreeCAD (IfcOpenShell)
- Allplan
- Tekla BIMsight

ezdxf.addons.meshex.**export_ifcZIP**(*filename: str | PathLike*, *mesh: MeshBuilder*, *entity_type=IfcEntityType.POLYGON_FACE_SET*, **, layer: str = 'MeshExport'*, *color: tuple[float, float, float] = (1.0, 1.0, 1.0)*)

Export the given *mesh* as zip-compressed IFC4 file. The filename suffix should be `.ifcZIP`. For more information see function *ifc4_dumps()*.

Parameters

- **filename** – zip filename, the data file has the same name with suffix `.ifc`
- **mesh** – *MeshBuilder*
- **entity_type** – *IfcEntityType*
- **layer** – layer name as string
- **color** – entity color as RGB tuple, values in the range [0,1]

Raises

IOError – IO error when opening the zip-file for writing

class ezdxf.addons.meshex.**IfcEntityType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

POLYGON_FACE_SET

"SurfaceModel" representation usable for open or closed surfaces.

CLOSED_SHELL

"Brep" representation usable for closed surfaces.

OPEN_SHELL

"SurfaceModel" representation usable for open surfaces.

6.12.16 OpenSCAD

Interface to the OpenSCAD application to apply boolean operations to *MeshBuilder* objects. For more information about boolean operations read the documentation of OpenSCAD. The OpenSCAD application is not bundled with *ezdxf*, you need to install the application yourself.

On Windows the path to the `openscad.exe` executable is stored in the config file (see *ezdxf.options*) in the "openscad-addon" section as key "win_exec_path", the default entry is:

```
[openscad-addon]
win_exec_path = "C:\Program Files\OpenSCAD\openscad.exe"
```

On Linux and macOS the `openscad` command is located by the `shutil.which()` function.

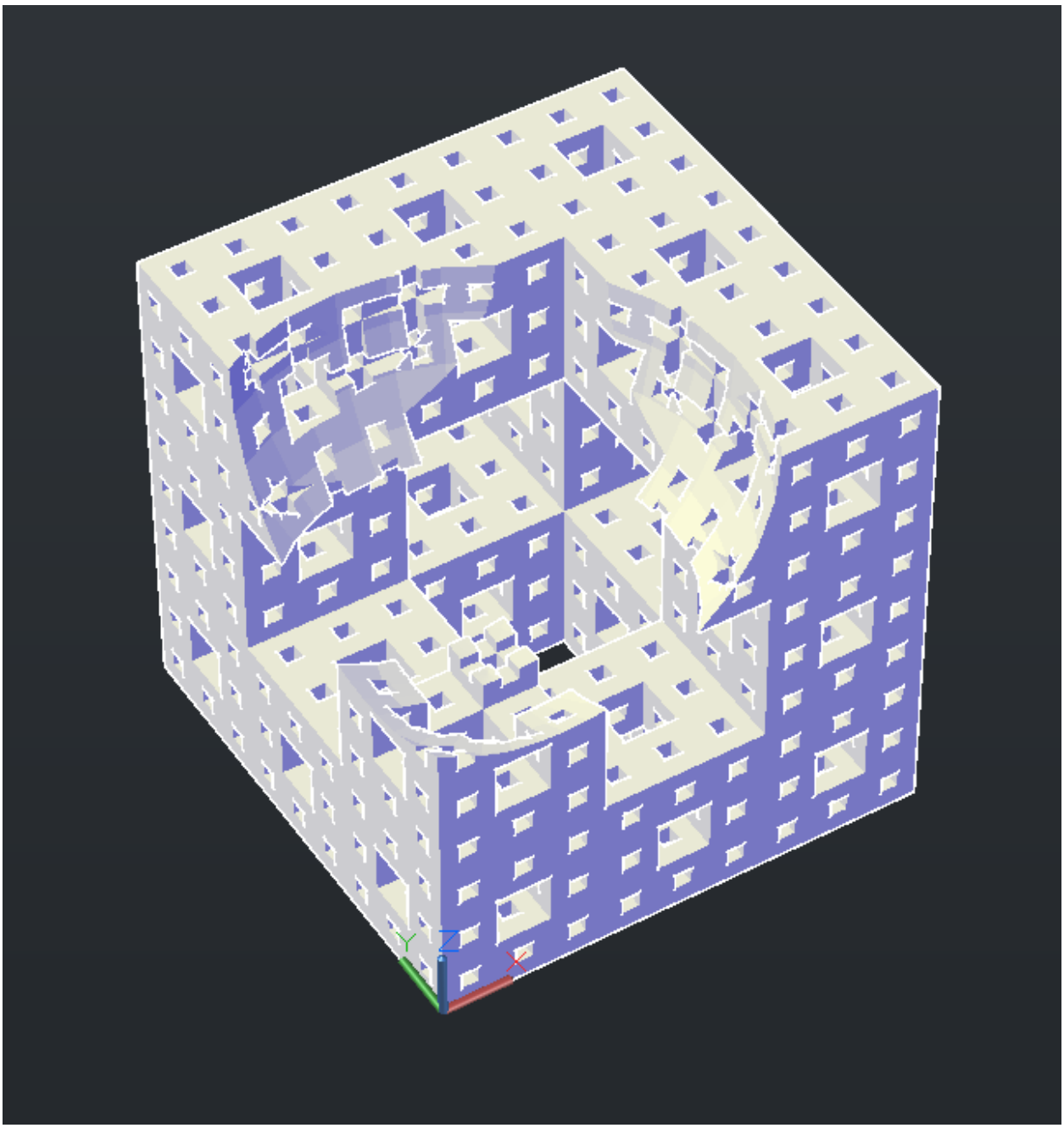
Example:

```
import ezdxf
from ezdxf.render import forms
from ezdxf.addons import MengerSponge, openscad

doc = ezdxf.new()
msp = doc.modelspace()

# 1. create the meshes:
sponge = MengerSponge(level=3).mesh()
sponge.flip_normals()  # important for OpenSCAD
sphere = forms.sphere(
    count=32, stacks=16, radius=0.5, quads=True
).translate(0.25, 0.25, 1)
sphere.flip_normals()  # important for OpenSCAD

# 2. create the script:
script = openscad.boolean_operation(openscad.DIFFERENCE, sponge, sphere)
```

(continues on next page)

(continued from previous page)

```
# 3. execute the script by OpenSCAD:
result = openscad.run(script)

# 4. render the MESH entity:
result.render_mesh(msp)

doc.set_modelspace_vport(6, center=(5, 0))
doc.saveas("OpenSCAD.dxf")
```



Functions

ezdxf.addons.openscad.**run**(*script: str*, *exec_path: str | None = None*) → MeshTransformer

Executes the given *script* by OpenSCAD and returns the result mesh as *MeshTransformer*.

Parameters

- **script** – the OpenSCAD script as string
- **exec_path** – path to the executable as string or None to use the default installation path

ezdxf.addons.openscad.**boolean_operation**(*op: Operation*, *mesh1: MeshBuilder*, *mesh2: MeshBuilder*) → str

Returns an OpenSCAD script to apply the given boolean operation to the given meshes.

The supported operations are:

- UNION
- DIFFERENCE
- INTERSECTION

ezdxf.addons.openscad.**is_installed**() → bool

Returns True if OpenSCAD is installed. On Windows only the default install path 'C:\Program Files\OpenSCAD\openscad.exe' is checked.

Script Class

class ezdxf.addons.openscad.**Script**

Helper class to build OpenSCAD scripts. This is a very simple string building class and does no checks at all! If you need more advanced features to build OpenSCAD scripts look at the packages pysolid and openpyscad.

add(*data: str*) → None

Add a string.

add_mirror(*v:* UVec) → None

Add a mirror() operation.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#mirror

Parameters

v – the normal vector of a plane intersecting the origin through which to mirror the object

add_multmatrix(*m:* Matrix44) → None

Add a transformation matrix of type *Matrix44* as multmatrix() operation.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#multmatrix

add_polyhedron(*mesh: MeshBuilder*) → None

Add *mesh* as polyhedron() command.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Primitive_Solids#polyhedron

add_polygon(*path: Iterable[*UVec*]*, *holes: Sequence[Iterable[*UVec*]] | None = None*) → None

Add a polygon() command. This is a 2D command, all z-axis values of the input vertices are ignored and all paths and holes are closed automatically.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Using_the_2D_Subsystem#polygon

Parameters

- **path** – exterior path
- **holes** – a sequence of one or more holes as vertices, or `None` for no holes

add_resize(*nx: float*, *ny: float*, *nz: float*, *auto: bool | tuple[bool, bool, bool] | None = None*) → None

Add a `resize()` operation.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#resize

Parameters

- **nx** – new size in x-axis
- **ny** – new size in y-axis
- **nz** – new size in z-axis
- **auto** – If the *auto* argument is set to `True`, the operation auto-scales any 0-dimensions to match. Set the *auto* argument as a 3-tuple of bool values to auto-scale individual axis.

add_rotate(*ax: float*, *ay: float*, *az: float*) → None

Add a `rotation()` operation.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#rotate

Parameters

- **ax** – rotation about the x-axis in degrees
- **ay** – rotation about the y-axis in degrees
- **az** – rotation about the z-axis in degrees

add_rotate_about_axis(*a: float*, *v:* UVec) → None

Add a `rotation()` operation about the given axis *v*.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#rotate

Parameters

- **a** – rotation angle about axis *v* in degrees
- **v** – rotation axis as *ezdxf.math.UVec* object

add_scale(*sx: float*, *sy: float*, *sz: float*) → None

Add a `scale()` operation.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#scale

Parameters

- **sx** – scaling factor for the x-axis
- **sy** – scaling factor for the y-axis
- **sz** – scaling factor for the z-axis

add_translate(*v:* UVec) → None

Add a `translate()` operation.

OpenSCAD docs: https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#translate

Parameters

v – translation vector

get_string() → str
Returns the OpenSCAD build script.

Boolean Operation Constants

ezdxf.addons.openscad.**UNION**

ezdxf.addons.openscad.**DIFFERENCE**

ezdxf.addons.openscad.**INTERSECTION**

openpyscad

This add-on is not a complete wrapper around OpenSCAD, if you need such a tool look at the openpyscad or pysolid packages at PyPI.

Not sure if the openpyscad package is still maintained, the last commit at github is more than a year old and did not pass the CI process! (state June 2022)

This code snippet shows how to get the *MeshTransformer* object from the basic openpyscad example:

```
from ezdxf.addons import openscad
import openpyscad as ops

c1 = ops.Cube([10, 20, 10])
c2 = ops.Cube([20, 10, 10])

# dump OpenSCAD script as string:
script = (c1 + c2).dumps()

# execute script and load the result as MeshTransformer instance:
mesh = openscad.run(script)
```

Create an openpyscad `Polyhedron` object from an *ezdxf* *MeshBuilder* object:

```
from ezdxf.render import forms
import openpyscad as ops

# create an ezdxf MeshBuilder() object
sphere = forms.sphere()
sphere.flip_normals()  # required for OpenSCAD

# create an openpyscad Polyhedron() object
polyhedron = ops.Polyhedron(
    points=[list(p) for p in sphere.vertices],  # convert Vec3 objects to lists!
    faces=[list(f) for f in sphere.faces],  # convert face tuples to face lists!
)

# create the OpenSCAD script:
script = polyhedron.dumps()
```

The type conversion is needed to get valid OpenSCAD code from openpyscad!

pysolid

The pysolid package seems to be better maintained than the openpyscad package, but this is just an opinion based on newer commits at github (link) for the pysolid package.

Same example for pysolid:

```
from ezdxf.addons import openscad
from solid import cube, render_scad

c1 = cube([10, 20, 10])
c2 = cube([20, 10, 10])

# dump OpenSCAD script as string:
script = render_scad(c1 + c2)

# execute script and load the result as MeshTransformer instance:
mesh = openscad.run(script)
```

Create a pysolid `polyhedron` object from an *ezdxf* *MeshBuilder* object:

```
from ezdxf.render import forms
from solid import polyhedron, scad_render

# create an ezdxf MeshBuilder() object
sphere = forms.sphere()
sphere.flip_normals()  # required for OpenSCAD

# create a pysolid polyhedron() object
ph = polyhedron(
    points=[v.xyz for v in sphere.vertices],  # convert Vec3 objects to tuples!
    faces=sphere.faces,  # types are compatible
)

# create the OpenSCAD script:
script = scad_render(ph)
```

6.12.17 TablePainter

This is an add-on for drawing tables build from DXF primitives.

This add-on was created for porting `dxfwrite` projects to `ezdxf` and was not officially documented for `ezdxf` versions prior the 1.0 release. For the 1.0 version of `ezdxf`, this class was added as an officially documented add-on because full support for the ACAD_TABLE entity is very unlikely due to the enormous complexity for both the entity itself, and for the required infrastructure and also the lack of a usable documentation to implement all that features.

Important: This add-on is not related to the ACAD_TABLE entity at all and and does not create ACAD_TABLE entities!

The table cells can contain multi-line text or BLOCK references. You can create your own cell types by extending the *CustomCell* class. The cells are addressed by zero-based row and column indices. A table cell can span over multiple columns and/or rows.

A *TextCell* can contain multi-line text with an arbitrary rotation angle or letters stacked from top to bottom. The *MTextSurrogate* add-on is used to create multi-line text compatible to DXF version R12.

A *BlockCell* contains block references (INSERT entities), if the block definition contains attribute definitions as ATTDEF entities, these attributes can be added automatically to the block reference as ATTRIB entities.

Note: The DXF format does not support clipping boxes ot paths, therefore the render method of any cell can render beyond the borders of the cell!

Tutorial

Set up a new DXF document:

```
import ezdxf
from ezdxf.enums import MTextEntityAlignment
from ezdxf.addons import TablePainter

doc = ezdxf.new("R2000")  # required for lineweight support
doc.header["$LWDISPLAY"] = 1  # show lineweights
doc.styles.add("HEAD", font="OpenSans-ExtraBold.ttf")
doc.styles.add("CELL", font="OpenSans-Regular.ttf")
```

Create a new *TablePainter* object with four rows and four columns, the insert location is the default render location but can be overriden in the `render()` method:

```
table = TablePainter(
    insert=(0, 0), nrows=4, ncols=4, cell_width=6.0, cell_height=2.0
)
```

Create a new *CellStyle* object for the table-header called “head”:

```
table.new_cell_style(
    "head",
    text_style="HEAD",
    text_color=ezdxf.colors.BLUE,
    char_height=0.7,
    bg_color=ezdxf.colors.LIGHT_GRAY,
    align=MTextEntityAlignment.MIDDLE_CENTER,
)
```

Redefine the default *CellStyle* for the content cells:

```
# reset default cell style
default_style = table.get_cell_style("default")
default_style.text_style = "CELL"
default_style.char_height = 0.5
default_style.align = MTextEntityAlignment.BOTTOM_LEFT
```

Set the table-header content:

```
for col in range(4):
    table.text_cell(0, col, f"Head[{col}]", style="head")
```

Set the cell content:

```
for row in range(1, 4):
    for col in range(4):
```

(continues on next page)

(continued from previous page)

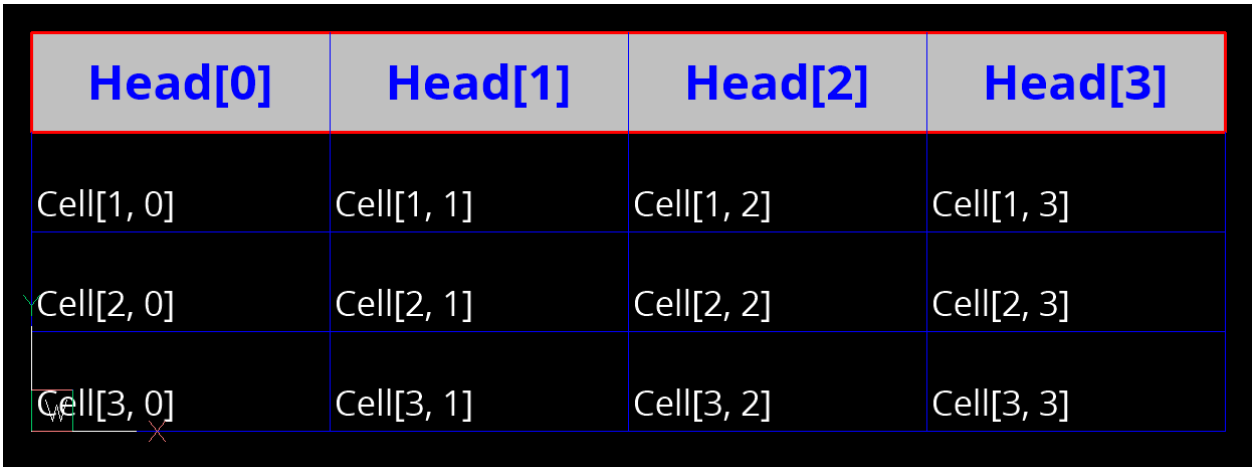
```
        # cell style is "default"
        table.text_cell(row, col, f"Cell[{row}, {col}]")
```

Add a red frame around the table-header:

```
# new cell style is required
red_frame = table.new_cell_style("red-frame")
red_borderline = table.new_border_style(color=ezdxf.colors.RED, lineweight=35)
# set the red borderline style for all cell borders
red_frame.set_border_style(red_borderline)
# create the frame object
table.frame(0, 0, 4, style="red-frame")
```

Render the table into the modelspace and export the DXF file:

```
# render the table, shifting the left-bottom of the table to the origin:
table.render(doc.modelspace(), insert=(0, table.table_height))

th = table.table_height
tw = table.table_width
doc.set_modelspace_vport(height=th * 1.5, center=(tw/2, th/2))
doc.saveas("table_tutorial.dxf")
```



See also:

- Example script: table_painter_addon.py

TablePainter

class ezdxf.addons.tablepainter.**TablePainter**(*insert: UVec*, *nrows: int*, *ncols: int*, *cell_width=DEFAULT_CELL_WIDTH*, *cell_height=DEFAULT_CELL_HEIGHT*, *default_grid=True*)

The TablePainter class renders tables build from DXF primitives.

The TablePainter instance contains all the data cells.

Parameters

- **insert** – insert location as or *UVec*

- **nrows** – row count
- **ncols** – column count
- **cell_width** – default cell width in drawing units
- **cell_height** – default cell height in drawing units
- **default_grid** – draw a grid of solid lines if `True`, otherwise draw only explicit defined borders, the default grid has a priority of 50.

bg_layer_name: str

background layer name, layer for the background SOLID entities, default is "TABLEBACKGROUND"

fg_layer_name: str

foreground layer name, layer for the cell content, default is "TABLECONTENT"

grid_layer_name: str

table grid layer name, layer for the cell border lines, default is "TABLEGRID"

property table_width: float

Returns the total table width.

property table_height: float

Returns the total table height.

set_col_width(*index: int*, *value: float*)

Set column width in drawing units of the given column index.

Parameters

- **index** – zero based column index
- **value** – new column width in drawing units

set_row_height(*index: int*, *value: float*)

Set row height in drawing units of the given row index.

Parameters

- **index** – zero based row index
- **value** – new row height in drawing units

text_cell(*row: int*, *col: int*, *text: str*, *span: tuple[int, int] = (1, 1)*, *style='default'*) → *TextCell*

Factory method to create a new text cell at location (row, col), with *text* as content, the *text* can be a line breaks `'\n'`. The final cell can spread over several cells defined by the argument *span*.

block_cell(*row: int*, *col: int*, *blockdef:* BlockLayout, *span: tuple[int, int] = (1, 1)*, *attribs=None*, *style='default'*) → *BlockCell*

Factory method to Create a new block cell at position (row, col).

Content is a block reference inserted by an INSERT entity, attributes will be added if the block definition contains ATTDEF. Assignments are defined by attribs-key to attdef-tag association.

Example: attribs = {'num': 1} if an ATTDEF with tag=='num' in the block definition exists, an attrib with text=str(1) will be created and added to the insert entity.

The cell spans over 'span' cells and has the cell style with the name 'style'.

set_cell(*row: int*, *col: int*, *cell: T*) → T

Insert a cell at position (row, col).

get_cell(*row: int*, *col: int*) → *Cell*

Get cell at location (row, col).

new_cell_style(*name: str*, ***kwargs*) → *CellStyle*

Factory method to create a new `CellStyle` object, overwrites an already existing cell style.

Parameters

- **name** – style name as string
- **kwargs** – see attributes of class `CellStyle`

get_cell_style(*name: str*) → *CellStyle*

Get cell style by name.

static new_border_style(*color: int = const.BYLAYER*, *status=True*, *priority: int = 100*, *linetype: str = 'BYLAYER'*, *lineweight: int = const.LINEWEIGHT_BYLAYER*) → *BorderStyle*

Factory method to create a new border style.

Parameters

- **status** – `True` for visible, `False` for invisible
- **color** – *AutoCAD Color Index (ACI)*
- **linetype** – linetype name, default is "BYLAYER"
- **lineweight** – lineweight as int, default is by layer
- **priority** – drawing priority, higher priorities cover lower priorities

frame(*row: int*, *col: int*, *width: int = 1*, *height: int = 1*, *style='default'*) → Frame

Creates a frame around the give cell area, starting at (row, col) and covering *width* columns and *height* rows. The *style* argument is the name of a `CellStyle`.

render(*layout: GenericLayoutType*, *insert:* UVec | *None = None*)

Render table to layout.

Cell

class ezdxf.addons.tablepainter.**Cell**

Abstract base class for table cells.

TextCell

class ezdxf.addons.tablepainter.**TextCell**

Implements a cell type containing a multi-line text. Uses the `MTextSurrogate` add-on to render the multi-line text, therefore the content of these cells is compatible to DXF R12.

Important: Use the factory method `TablePainter.text_cell()` to instantiate text cells.

BlockCell

class ezdxf.addons.tablepainter.**BlockCell**(*table:* TablePainter, *blockdef:* BlockLayout, *style='default'*, *attribs=None*, *span: tuple[int, int] = (1, 1)*)

Implements a cell type containing a block reference.

Parameters

- **table** – table object
- **blockdef** – *ezdxf.layouts.BlockLayout* instance
- **attribs** – BLOCK attributes as (tag, value) dictionary
- **style** – cell style name as string
- **span** – tuple(rows, cols) area of cells to cover

Implements a cell type containing a block reference.

Important: Use the factory method *TablePainter.block_cell()* to instantiate block cells.

CustomCell

class ezdxf.addons.tablepainter.**CustomCell**

Base class to implement custom cells. Overwrite the *render()* method to render the cell. The custom cell type has to be instantiated by the user and added to the table by the *TablePainter.set_cell()* method.

render(*layout: GenericLayoutType*, *coords: Sequence[float]*, *layer: str*)

Renders the cell content into the given *layout*.

The render space is defined by the argument *coords* which is a tuple of 4 float values in the order: left, right, top, bottom. These values are layout coordinates in drawing units. The DXF format does not support clipping boxes, therefore the render method can render beyond these borders!

CellStyle

class ezdxf.addons.tablepainter.**CellStyle**(*data: dict[str, Any] | None = None*)

Cell style object.

Important: Always instantiate new styles by the factory method: *TablePainter.new_cell_style()*

text_style: str

Textstyle name as string, ignored by *BlockCell*

char_height: float

text height in drawing units, ignored by *BlockCell*

line_spacing: float

line spacing in percent, distance of line base points = `char_height * line_spacing`, ignored by *BlockCell*

scale_x: float
text stretching factor (width factor) or block reference x-scaling factor

scale_y: float
block reference y-scaling factor, ignored by *TextCell*

text_color: int
AutoCAD Color Index (ACI) for text, ignored by *BlockCell*

rotation: float
text or block rotation in degrees

stacked: bool
Stacks letters of *TextCell* instances from top to bottom without rotating the characters if `True`, ignored by *BlockCell*

align: MTextEntityAlignment
text and block alignment, see *ezdxf.enums.MTextEntityAlignment*

margin_x: float
left and right cell margin in drawing units

margin_y: float
top and bottom cell margin in drawing units

bg_color: int
cell background color as *AutoCAD Color Index (ACI)*, ignored by *BlockCell*

left: BorderStyle
left cell border style

top: BorderStyle
top cell border style

right: BorderStyle
right cell border style

bottom: BorderStyle
bottom cell border style

set_border_status(*left=True*, *right=True*, *top=True*, *bottom=True*)
Set status of all cell borders at once.

set_border_style(*style:* BorderStyle, *left=True*, *right=True*, *top=True*, *bottom=True*)
Set border styles of all cell borders at once.

static get_default_border_style() → *BorderStyle*

BorderStyle

class ezdxf.addons.tablepainter.**BorderStyle**(*status: bool = DEFAULT_BORDER_STATUS*, *color: int = DEFAULT_BORDER_COLOR*, *linetype: str = DEFAULT_BORDER_LINETYPE*, *lineweight=const.LINEWEIGHT_BYLAYER*, *priority: int = DEFAULT_BORDER_PRIORITY*)

Border style class.

Important: Always instantiate new border styles by the factory method: *TablePainter.new_border_style()*

status: bool
border status, `True` for visible, `False` for hidden

color: int
AutoCAD Color Index (ACI)

linetype: str
linetype name as string, default is "BYLAYER"

lineweight: int
lineweight as int, default is by layer

priority: int
drawing priority, higher values cover lower values

6.12.18 MTextSurrogate for DXF R12

class ezdxf.addons.**MTextSurrogate**(*text: str*, *insert:* UVec, *line_spacing: float = 1.5*, *align=MTextEntityAlignment.TOP_LEFT*, *char_height: float = 1.0*, *style='STANDARD'*, *oblique: float = 0.0*, *rotation: float = 0.0*, *width_factor: float = 1.0*, *mirror=Mirror.NONE*, *layer='0'*, *color: int = const.BYLAYER*)

MTEXT surrogate for DXF R12 build up by TEXT Entities. This add-on was added to simplify the transition from `dxfwrite` to `ezdxf`.

The rich-text formatting capabilities for the regular MTEXT entity are not supported, if these features are required use the regular MTEXT entity and the *MTextExplode* add-on to explode the MTEXT entity into simpler DXF primitives.

Important: The align-point is always the insert-point, there is no need for a second align-point because the horizontal alignments FIT, ALIGN, BASELINE_MIDDLE are not supported.

Parameters

- **text** – content as string
- **insert** – insert location in drawing units
- **line_spacing** – line spacing in percent of height, 1.5 = 150% = 1+1/2 lines
- **align** – text alignment as *MTextEntityAlignment* enum

- **char_height** – text height in drawing units
- **style** – *Textstyle* name as string
- **oblique** – oblique angle in degrees, where 0 is vertical
- **rotation** – text rotation angle in degrees
- **width_factor** – text width factor as float
- **mirror** – `MTextSurrogate.MIRROR_X` to mirror the text horizontal or `MTextSurrogate.MIRROR_Y` to mirror the text vertical
- **layer** – layer name as string
- **color** – *AutoCAD Color Index (ACI)*

render (*layout: GenericLayoutType*) → None

Render the multi-line content as separated TEXT entities into the given *layout* instance.

6.12.19 ASTM-D6673-10 Exporter

This add-on creates special DXF files for use by Gerber Technology applications which have a low quality DXF parser and cannot parse/ignore BLOCKS which do not contain data according the ASTM-D6673-10 standard. The function *export_file()* exports DXF R12 and only DXF R12 files which do not contain the default "$MODEL_SPACE" and "$PAPER_SPACE" layout block definitions, have an empty HEADER section and no TABLES section. These special requirements of the Gerber Technology parser are annoying, but correspond to the DXF R12 standard.

Autodesk applications maybe complain about invalid BLOCK names such as "Shape 0_M", which in my opinion are valid, maybe spaces were not allowed in the original R12 version, but this is just a minor issue and is more a problem of the picky Autodesk DXF parser, which is otherwise very forgiving for DXF R12 files.

```
import ezdxf
from ezdxf.addons import gerber_D6673

doc = ezdxf.new("R12")  # the export function rejects other DXF versions
msp = doc.modelspace()

# Create your content according the ASTM-D6673-10 standard
# Do not use any linetypes or text styles, the TABLES section will not be exported.
# The ASTM-D6673-10 standard supports only 7-bit ASCII characters.

gerber_D6673.export_file(doc, "gerber_file.dxf")
```

ezdxf.addons.gerber_D6673.**export_file** (*doc:* Drawing, *filename: str | PathLike*) → None

Exports the specified DXF R12 document, which should contain content conforming to the ASTM-D6673-10 standard, in a special way so that Gerber Technology applications can parse it by their low-quality DXF parser.

ezdxf.addons.gerber_D6673.**export_stream** (*doc:* Drawing, *stream: TextIO*) → None

Exports the specified DXF R12 document into a *stream* object.

6.13 DXF Internals

- DXF Reference provided by Autodesk.
- DXF Developer Documentation provided by Autodesk.

6.13.1 Basic DXF Structures

DXF File Encoding

DXF R2004 and prior

Drawing files of DXF R2004 (AC1018) and prior are saved as ASCII files with the encoding set by the header variable $DWGCODEPAGE, which is `ANSI_1252` by default if $DWGCODEPAGE is not set.

Characters used in the drawing which do not exist in the chosen ASCII encoding are encoded as unicode characters with the schema `\U+nnnn`. see Unicode table

Known $DWGCODEPAGE encodings

DXF	Python	Name
ANSI_874	cp874	Thai
ANSI_932	cp932	Japanese
ANSI_936	gbk	UnifiedChinese
ANSI_949	cp949	Korean
ANSI_950	cp950	TradChinese
ANSI_1250	cp1250	CentralEurope
ANSI_1251	cp1251	Cyrillic
ANSI_1252	cp1252	WesternEurope
ANSI_1253	cp1253	Greek
ANSI_1254	cp1254	Turkish
ANSI_1255	cp1255	Hebrew
ANSI_1256	cp1256	Arabic
ANSI_1257	cp1257	Baltic
ANSI_1258	cp1258	Vietnam

DXF R2007 and later

Starting with DXF R2007 (AC1021) the drawing file is UTF-8 encoded, the header variable $DWGCODEPAGE is still in use, but I don't know, if the setting still has any meaning.

Encoding characters in the unicode schema `\U+nnnn` is still functional.

See also:

String value encoding

DXF Tags

A Drawing Interchange File is simply an ASCII text file with a file type of .dxf and special formatted text. The basic file structure are DXF tags, a DXF tag consist of a DXF group code as an integer value on its own line and a the DXF value on the following line. In the ezdxf documentation DXF tags will be written as (group code, value).

With the introduction of extended symbol names in DXF R2000, the 255-character limit for strings has been increased to 2049 single-byte characters not including the newline at the end of the line. Nonetheless its safer to use only strings with 255 and less characters, because its not clear if this fact is true for ALL string group codes or only for symbols like layer- or text style names and not all 3rd party libraries may handle this fact correct. The MTEXT content and binary data is still divided into chunks with less than 255 characters.

Group codes are indicating the value type:

Group Code	Value Type
0-9	String
10-39	Double precision 3D point value
40-59	Double-precision floating-point value
60-79	16-bit integer value
90-99	32-bit integer value
100	String
102	String
105	String representing hexadecimal (hex) handle value
110-119	Double precision floating-point value
120-129	Double precision floating-point value
130-139	Double precision floating-point value
140-149	Double precision scalar floating-point value
160-169	64-bit integer value
170-179	16-bit integer value
210-239	Double-precision floating-point value
270-279	16-bit integer value
280-289	16-bit integer value
290-299	Boolean flag value
300-309	Arbitrary text string
310-319	String representing hex value of binary chunk
320-329	Arbitrary pointer, hex object ID, not translated during INSERT and XREF operations
330-339	Soft-pointer, hex object ID, translated during INSERT and XREF operations
340-349	Hard-pointer, hex object ID, translated during INSERT and XREF operations
350-359	Soft-owner, hex object ID, translated during INSERT and XREF operations
360-369	Hard-owner, hex object ID, translated during INSERT and XREF operations
370-379	16-bit integer value
380-389	16-bit integer value
390-399	String representing hex handle value
400-409	16-bit integer value
410-419	String
420-429	32-bit integer value
430-439	String
440-449	32-bit integer value
450-459	Long
460-469	Double-precision floating-point value
470-479	String
480-481	Hard-pointer, hex object ID, translated during INSERT and XREF operations
999	Comment (string)

continues on next page

Table 4 – continued from previous page

Group Code	Value Type
1000-1009	String
1010-1059	Double-precision floating-point value
1060-1070	16-bit integer value
1071	32-bit integer value

Explanation for some important group codes:

Group Code	Meaning
0	DXF structure tag, entity start/end or table entries
1	The primary text value for an entity
2	A name: Attribute tag, Block name, and so on. Also used to identify a DXF section or table name.
3-4	Other textual or name values
5	Entity handle as hex string (fixed)
6	Line type name (fixed)
7	Text style name (fixed)
8	Layer name (fixed)
9	Variable name identifier (used only in HEADER section of the DXF file)
10	Primary X coordinate (start point of a Line or Text entity, center of a Circle, etc.)
11-18	Other X coordinates
20	Primary Y coordinate. 2n values always correspond to 1n values and immediately follow them in the file (expected by ez
21-28	Other Y coordinates
30	Primary Z coordinate. 3n values always correspond to 1n and 2n values and immediately follow them in the file (expecte
31-38	Other Z coordinates
39	This entity's thickness if nonzero (fixed)
40-48	Float values (text height, scale factors, etc.)
49	Repeated value - multiple 49 groups may appear in one entity for variable length tables (such as the dash lengths in the I
50-58	Angles in degree
62	Color number (fixed)
66	"Entities follow" flag (fixed), only in INSERT and POLYLINE entities
67	Identifies whether entity is in modelspace (0) or paperspace (1)
68	Identifies whether viewport is on but fully off screen, is not active, or is off
69	Viewport identification number
70-78	Integer values such as repeat counts, flag bits, or modes
105	DIMSTYLE entity handle as hex string (fixed)
210, 220, 230	X, Y, and Z components of extrusion direction (fixed)
310	Proxy entity graphics as binary encoded data
330	Owner handle as hex string
347	MATERIAL handle as hex string
348	VISUALSTYLE handle as hex string
370	Lineweight in mm times 100 (e.g. 0.13mm = 13).
390	PLOTSTYLE handle as hex string
420	True color value as 0x00RRGGBB 24-bit value
430	Color name as string
440	Transparency value 0x020000TT 0 = fully transparent / 255 = opaque
999	Comments

For explanation of all group codes see: DXF Group Codes in Numerical Order Reference provided by Autodesk

Extended Data

DXF R2018 Reference

Extended data (XDATA) is created by AutoLISP or ObjectARX applications but any other application like *ezdxf* can also define XDATA. If an entity contains extended data, it **follows** the entity's normal definition.

But extended group codes (>=1000) can appear **before** the XDATA section, an example is the BLOCKBASEPOINTPARAMETER entity in AutoCAD Civil 3D or AutoCAD Map 3D.

Group Code	Description
1000	Strings in extended data can be up to 255 bytes long (with the 256th byte reserved for the null character)
1001	(fixed) Registered application name (ASCII string up to 31 bytes long) for XDATA
1002	(fixed) An extended data control string can be either '{' or '}'. These braces enable applications to organize their data by subdividing the data into lists. Lists can be nested.
1003	Name of the layer associated with the extended data
1004	Binary data is organized into variable-length chunks. The maximum length of each chunk is 127 bytes. In ASCII DXF files, binary data is represented as a string of hexadecimal digits, two per binary byte
1005	Database Handle of entities in the drawing database, see also: *About 1005 Group Codes*
1010, 1020, 1030	Three real values, in the order X, Y, Z. They can be used as a point or vector record that will not be modified at any transformation of the entity.
1011, 1021, 1031	a WCS point that is moved, scaled, rotated and mirrored along with the entity
1012, 1012, 1022	a WCS displacement that is scaled, rotated and mirrored along with the entity, but is not moved
1013, 1023, 1033	a WCS direction that is rotated and mirrored along with the entity, but is not moved or scaled
1040	A real value
1041	Distance, a real value that is scaled along with the parent entity
1042	Scale Factor, also a real value that is scaled along with the parent. The difference between a distance and a scale factor is application-defined
1070	A 16-bit integer (signed or unsigned)
1071	A 32-bit signed (long) integer

The (1001, …) tag indicates the beginning of extended data. In contrast to normal entity data, with extended data the same group code can appear multiple times, and **order is important**.

Extended data is grouped by registered application name. Each registered application group begins with a (1001, APPID) tag, with the application name as APPID string value. Registered application names correspond to APPID symbol table entries.

An application can use as many APPID names as needed. APPID names are permanent, although they can be purged if they aren't currently used in the drawing. Each APPID name can have **no more than one data group** attached to each entity. Within an application group, the sequence of extended data groups and their meaning is defined by the application.

String value encoding

String values stored in a DXF file is plain ASCII or UTF-8, AutoCAD also supports CIF (Common Interchange Format) and MIF (Maker Interchange Format) encoding. The UTF-8 format is only supported in DXF R2007 and later.

Ezdxf on import converts all strings into Python unicode strings without encoding or decoding CIF/MIF.

String values containing Unicode characters are represented with control character sequences `\U+nnnn`. (e.g. `r'TEST\U+7F3A\U+4E4F\U+89E3\U+91CA\U+6B63THIS\U+56FE'`)

To support the DXF unicode encoding ezdxf registers an encoding codec *dxf_backslash_replace*, defined in `ezdxf.lldxf.encoding()`.

String values can be stored with these dxf group codes:

- 0 - 9
- 100 - 101
- 300 - 309
- 410 - 419
- 430 - 439
- 470 - 479
- 999 - 1003

Multi tag text (MTEXT)

If the text string is less than 250 characters, all characters appear in tag (1, …). If the text string is longer than 250 characters, the string is divided into 250-character chunks, which appear in one or more (3, …) tags. If (3, …) tags are used, the last group is a (1, …) tag and has fewer than 250 characters:

```
3
... TwoHundredAndFifty Characters ....
3
... TwoHundredAndFifty Characters ....
1
less than TwoHundredAndFifty Characters
```

As far I know this is only supported by the MTEXT entity.

See also:

DXF File Encoding

DXF R13 and later tag structure

With the introduction of DXF R13 Autodesk added additional group codes and DXF tag structures to the DXF Standard.

Subclass Markers

Subclass markers (100, Subclass Name) divides DXF objects into several sections. Group codes can be reused in different sections. A subclass ends with the following subclass marker or at the beginning of xdata or the end of the object. See Subclass Marker Example in the DXF Reference.

Quote about group codes from the DXF reference

> Some group codes that define an entity always appear; others are optional and appear only if their values differ from the defaults.
>
> **Do not** write programs that **rely on the order given here**. The end of an entity is indicated by the next 0 group, which begins the next entity or indicates the end of the section.
>
> **Note:** Accommodating DXF files from future releases of AutoCAD will be easier if you write your DXF processing program in a table-driven way, ignore undefined group codes, and make no assumptions about the order of group codes in an entity. With each new AutoCAD release, new group codes will be added to entities to accommodate additional features.

Usage of group codes in subclasses twice

Some later entities entities contains the same group code twice for different purposes, so order in the sense of which one comes first is important. (e.g. ATTDEF group code 280)

Tag order is sometimes important especially for AutoCAD

In LWPOLYLINE the order of tags is important, if the *count* tag is not the first tag in the AcDbPolyline subclass, AutoCAD will not close the polyline when the *close* flag is set, by the way other applications like BricsCAD ignores the tag order and renders the polyline always correct.

Extension Dictionary

The extension dictionary is an optional sequence that stores the handle of a DICTIONARY object that belongs to the current object, which in turn may contain entries. This facility allows attachment of arbitrary database objects to any database object. Any object or entity may have this section.

The extension dictionary tag sequence:

```
102
{ACAD_XDICTIONARY
360
Hard-owner ID/handle to owner dictionary
102
}
```

Persistent Reactors

Persistent reactors are an optional sequence that stores object handles of objects registering themselves as reactors on the current object. Any object or entity may have this section.

The persistent reactors tag sequence:

```
102
{ACAD_REACTORS
330
first Soft-pointer ID/handle to owner dictionary
330
second Soft-pointer ID/handle to owner dictionary
...
102
}
```

Application-Defined Codes

Starting at DXF R13, DXF objects can contain application-defined codes outside of XDATA. This application-defined codes can contain any tag except (0, …) and (102, '{…'). "{YOURAPPID" means the APPID string with an preceding "{". The application defined data tag sequence:

```
102
{YOURAPPID
...
102
}
```

`(102, 'YOURAPPID}')` is also a valid closing tag:

```
102
{YOURAPPID
...
102
YOURAPPID}
```

All groups defined with a beginning (102, …) appear in the DXF reference before the first subclass marker, I don't know if these groups can appear after the first or any subclass marker. Ezdxf accepts them at any position, and by default ezdxf adds new app data in front of the first subclass marker to the first tag section of an DXF object.

Exception XRECORD: Tags with group code 102 and a value string without a preceding "{" or the scheme "YOURAPPID}", should be treated as usual group codes.

Embedded Objects

The concept of embedded objects was introduced with AutoCAD 2018 (DXF version AC1032) and this is the only information I found about it at the Autodesk knowledge base: Embedded and Encapsulated Objects

Quote from Embedded and Encapsulated Objects:

> For DXF filing, the embedded object must be filed out and in after all the data of the encapsulating object has been filed out and in.
>
> A separator is needed between the encapsulating object's data and the subsequent embedded object's data. The separator must be similar in function to the group 0 or 100 in that it must cause the filer to stop reading

data. The normal DXF group code 0 cannot be used because DXF proxies use it to determine when to stop reading data. The group code 100 could have been used, but it might have caused confusion when manually reading a DXF file, and there was a need to distinguish when an embedded object is about to be written out in order to do some internal bookkeeping. Therefore, the DXF group code 101 was introduced.

Hard facts:

- Only used in ATTRIB, ATTDEF (embedded MTEXT) and MTEXT (columns) in DXF R2018.
- Embedded object start with (101, "Embedded Object") tag
- Embedded object is appended to the encapsulated object
- Embedded object tags can contain any group code except the DXF structure tag (0, …)

Unconfirmed assumptions:

- The embedded object is written before the *Extended Data*. No examples for entities including embedded objects and XDATA at the same time.
- XDATA sections replaced by embedded objects, at least for the MTEXT entity
- The encapsulating object can contain more than one embedded object.
- Embedded objects separated by (101, "Embedded Object") tags
- every entity can contain embedded objects

Real world example from an AutoCAD 2018 file:

```
100       <<< start of encapsulating object
AcDbMText
10
2762.148
20
2327.073
30
0.0
40
2.5
41
18.852
46
0.0
71
1
72
5
1
{\fArial|b0|i0|c162|p34;CHANGE;\P\P\PTEXT}
73
1
44
1.0
101       <<< start of embedded object
Embedded Object
70
1
10
1.0
20
0.0
```

(continues on next page)

(continued from previous page)

```
30
0.0
11
2762.148
21
2327.073
31
0.0
40
18.852
41
0.0
42
15.428
43
15.043
71
2
72
1
44
18.852
45
12.5
73
0
74
0
46
0.0
```

Handles

A handle is an arbitrary but in your DXF file unique hex value as string like '10FF'. It is common to to use uppercase letters for hex numbers. Handle can have up to 16 hexadecimal digits (8 bytes).

For DXF R10 until R12 the usage of handles was optional. The header variable $HANDLING set to 1 indicate the usage of handles, else $HANDLING is 0 or missing.

For DXF R13 and later the usage of handles is mandatory and the header variable $HANDLING was removed.

The $HANDSEED variable in the header section should be greater than the biggest handle used in the DXF file, so a CAD application can assign handle values starting with the $HANDSEED value. But as always, don't rely on the header variable it could be wrong, AutoCAD ignores this value.

Handle Definition

Entity handle definition is always the `(5, ...)`, except for entities of the DIMSTYLE table `(105, ...)`, because the DIMSTYLE entity has also a group code 5 tag for DIMBLK.

Handle Pointer

A pointer is a reference to a DXF object in the same DXF file. There are four types of pointers:

- Soft-pointer handle
- Hard-pointer handle
- Soft-owner handle
- Hard-owner handle

Also, a group code range for "arbitrary" handles is defined to allow convenient storage of handle values that are unchanged at any operation (AutoCAD).

Pointer and Ownership

A pointer is a reference that indicates usage, but not possession or responsibility, for another object. A pointer reference means that the object uses the other object in some way, and shares access to it. An ownership reference means that an owner object is responsible for the objects for which it has an owner handle. An object can have any number of pointer references associated with it, but it can have only one owner.

Hard and Soft References

Hard references, whether they are pointer or owner, protect an object from being purged. Soft references do not.

In AutoCAD, block definitions and complex entities are hard owners of their elements. A symbol table and dictionaries are soft owners of their elements. Polyline entities are hard owners of their vertex and seqend entities. Insert entities are hard owners of their attrib and seqend entities.

When establishing a reference to another object, it is recommended that you think about whether the reference should protect an object from the PURGE command.

A hard- and soft pointers will be translated during INSERT and XREF operations.

Arbitrary Handles

Arbitrary handles are distinct in that they are not translated to session-persistent identifiers internally, or to entity names in AutoLISP, and so on. They are stored as handles. When handle values are translated in drawing-merge operations, arbitrary handles are ignored.

In all environments, arbitrary handles can be exchanged for entity names of the current drawing by means of the handent functions. A common usage of arbitrary handles is to refer to objects in external DXF and DWG files.

About 1005 Group Codes

`(1005, ...)` xdata have the same behavior and semantics as soft pointers, which means that they are translated whenever the host object is merged into a different drawing. However, 1005 items are not translated to session-persistent identifiers or internal entity names in AutoLISP and ObjectARX. They are stored as handles.

When a drawing with handles and extended data handles is imported into another drawing using INSERT, INSERT*, XREF Bind, XBIND, or partial OPEN, the extended data handles are **translated** in the same manner as their corresponding entity handles, thus maintaining their binding. This is also done in the EXPLODE block operation or for any other AutoCAD operation. When AUDIT detects an extended data handle that doesn't match the handle of an entity in the drawing file, it is considered an error. If AUDIT is fixing entities, it sets the handle to "0"

DXF File Structure

A DXF File is simply an ASCII text file with a file type of .dxf and special formatted text. The basic file structure are DXF tags, a DXF tag consist of a DXF group code as an integer value on its own line and a the DXF value on the following line. In the ezdxf documentation DXF tags will be written as `(group code, value)`. There exist a binary DXF format, but it seems that it is not often used and for reducing file size, zipping is much more efficient. *ezdxf* does support reading binary encoded DXF files.

See also:

For more information about DXF tags see: *DXF Tags*

A usual DXF file is organized in sections, starting with the DXF tag (0, 'SECTION') and ending with the DXF tag (0, 'ENDSEC'). The (0, 'EOF') tag signals the end of file.

1. **HEADER:** General information about the drawing is found in this section of the DXF file. Each parameter has a variable name starting with '$' and an associated value. Has to be the first section.
2. **CLASSES:** Holds the information for application defined classes. (DXF R13 and later)
3. **TABLES:**: Contains several tables for style and property definitions.
   - Linetype table (LTYPE)
   - Layer table (LAYER)
   - Text Style table (STYLE)
   - View table (VIEW): (IMHO) layout of the CAD working space, only interesting for interactive CAD applications
   - Viewport configuration table (VPORT): The VPORT table is unique in that it may contain several entries with the same name (indicating a multiple-viewport configuration). The entries corresponding to the active viewport configuration all have the name *ACTIVE. The first such entry describes the current viewport.
   - Dimension Style table (DIMSTYLE)
   - User Coordinate System table (UCS) (IMHO) only interesting for interactive CAD applications
   - Application Identification table (APPID): Table of names for all applications registered with a drawing.
   - Block Record table (BLOCK_RECORD) (DXF R13 and Later)
4. **BLOCKS:** Contains all block definitions. The block name *Model_Space or *MODEL_SPACE is reserved for the drawing modelspace and the block name *Paper_Space or *PAPER_SPACE is reserved for the *active* paperspace layout. Both block definitions are empty, the content of the modelspace and the *active* paperspace is stored in the ENTITIES section. The entities of other layouts are stored in special block definitions called *Paper_Spacennn, nnn is an arbitrary but unique number.

5. **ENTITIES:** Contains all graphical entities of the modelspace and the *active* paperspace layout. Entities of other layouts are stored in the BLOCKS sections.
6. **OBJECTS:** Contains all non-graphical objects of the drawing (DXF R13 and later)
7. **THUMBNAILIMAGE:** Contains a preview image of the DXF file, it is optional and can usually be ignored. (DXF R13 and later)
8. **ACDSDATA:** (DXF R2013 and later) No information in the DXF reference about this section
9. **END OF FILE**

For further information read the original DXF Reference.

Structure of a usual DXF R12 file:

```
0           <<< Begin HEADER section, has to be the first section
SECTION
2
HEADER
            <<< Header variable items go here
0           <<< End HEADER section
ENDSEC
0           <<< Begin TABLES section
SECTION
2
TABLES
0
TABLE
2
VPORT
70          <<< viewport table maximum item count
            <<< viewport table items go here
0
ENDTAB
0
TABLE
2
APPID, DIMSTYLE, LTYPE, LAYER, STYLE, UCS, VIEW, or VPORT
70          <<< Table maximum item count, a not reliable value and ignored by AutoCAD
            <<< Table items go here
0
ENDTAB
0           <<< End TABLES section
ENDSEC
0           <<< Begin BLOCKS section
SECTION
2
BLOCKS
            <<< Block definition entities go here
0           <<< End BLOCKS section
ENDSEC
0           <<< Begin ENTITIES section
SECTION
2
ENTITIES
            <<< Drawing entities go here
0           <<< End ENTITIES section
ENDSEC
0           <<< End of file marker (required)
```

(continues on next page)

(continued from previous page)

```
EOF
```

Minimal DXF Content

DXF R12

Contrary to the previous chapter, the DXF R12 format (AC1009) and prior requires just the ENTITIES section:

```
0
SECTION
2
ENTITIES
0
ENDSEC
0
EOF
```

DXF R13/R14 and later

DXF version R13/14 and later needs much more DXF content than DXF R12.

Required sections: HEADER, CLASSES, TABLES, ENTITIES, OBJECTS

The HEADER section requires two entries:

- $ACADVER
- $HANDSEED

The CLASSES section can be empty, but some DXF entities requires class definitions to work in AutoCAD.

The TABLES section requires following tables:

- VPORT entry *ACTIVE is not required! Empty table is ok for AutoCAD.
- LTYPE with at least the following line types defined:
  - BYBLOCK
  - BYLAYER
  - CONTINUOUS
- LAYER with at least an entry for layer '0'
- STYLE with at least an entry for style STANDARD
- VIEW can be empty
- UCS can be empty
- APPID with at least an entry for ACAD
- DIMSTYLE with at least an entry for style STANDARD
- BLOCK_RECORDS with two entries:
  - *MODEL_SPACE
  - *PAPER_SPACE

The BLOCKS section requires two BLOCKS:

- *MODEL_SPACE
- *PAPER_SPACE

The ENTITIES section can be empty.

The OBJECTS section requires following entities:

- DICTIONARY - the root dict - one entry named ACAD_GROUP
- DICTIONARY ACAD_GROUP can be empty

Minimal DXF to download: https://github.com/mozman/ezdxf/tree/master/examples_dxf

Data Model

Database Objects

(from the DXF Reference)

AutoCAD drawings consist largely of structured containers for database objects. Database objects each have the following features:

- A handle whose value is unique to the drawing/DXF file, and is constant for the lifetime of the drawing. This format has existed since AutoCAD Release 10, and as of AutoCAD Release 13, handles are always enabled.
- An optional XDATA table, as entities have had since AutoCAD Release 11.
- An optional persistent reactor table.
- An optional ownership pointer to an extension dictionary which, in turn, owns subobjects placed in it by an application.

Symbol tables and symbol table records are database objects and, thus, have a handle. They can also have xdata and persistent reactors in their DXF records.

DXF R12 Data Model

The DXF R12 data model is identical to the file structure:

- HEADER section: common settings for the DXF drawing
- TABLES section: definitions for LAYERS, LINETYPE, STYLES ….
- BLOCKS section: block definitions and its content
- ENTITIES section: modelspace and paperspace content

References are realized by simple names. The INSERT entity references the BLOCK definition by the BLOCK name, a TEXT entity defines the associated STYLE and LAYER by its name and so on, handles are not needed. Layout association of graphical entities in the ENTITIES section by the paper_space tag `(67, 0 or 1)`, 0 or missing tag means modelspace, 1 means paperspace. The content of BLOCK definitions is enclosed by the BLOCK and the ENDBLK entity, no additional references are needed.

A clean and simple file structure and data model, which seems to be the reason why the DXF R12 Reference (released 1992) is still a widely used file format and Autodesk/AutoCAD supports the format by reading and writing DXF R12 files until today (DXF R13/R14 has no writing support by AutoCAD!).

TODO: list of available entities

See also:

More information about the DXF *DXF File Structure*

DXF R13+ Data Model

With the DXF R13 file format, handles are mandatory and they are really used for organizing the new data structures introduced with DXF R13.

The HEADER section is still the same with just more available settings.

The new CLASSES section contains AutoCAD specific data, has to be written like AutoCAD it does, but must not be understood.

The TABLES section got a new BLOCK_RECORD table - see *Block Management Structures* for more information.

The BLOCKS sections is mostly the same, but with handles, owner tags and new ENTITY types. Not active paperspace layouts store their content also in the BLOCKS section - see *Layout Management Structures* for more information.

The ENTITIES section is also mostly same, but with handles, owner tags and new ENTITY types.

TODO: list of new available entities

And the new OBJECTS section - now its getting complicated!

Most information about the OBJECTS section is just guessed or gathered by trail and error, because the documentation of the OBJECTS section and its objects in the DXF reference provided by Autodesk is very shallow. This is also the reason why I started the DXF Internals section, may be it helps other developers to start one or two steps above level zero.

The OBJECTS sections stores all the non-graphical entities of the DXF drawing. Non-graphical entities from now on just called 'DXF objects' to differentiate them from graphical entities, just called 'entities'. The OBJECTS section follows commonly the ENTITIES section, but this is not mandatory.

DXF R13 introduces several new DXF objects, which resides exclusive in the OBJECTS section, taken from the DXF R14 reference, because I have no access to the DXF R13 reference, the DXF R13 reference is a compiled .hlp file which can't be read on Windows 10 or later, this a perfect example for not using closed (proprietary) data formats ;):

- DICTIONARY: a general structural entity as a <name: handle> container
- ACDBDICTIONARYWDFLT: a DICTIONARY with a default value
- DICTIONARYVAR: used by AutoCAD to store named values in the database
- ACAD_PROXY_OBJECT: proxy object for entities created by other applications than AutoCAD
- GROUP: groups graphical entities without the need of a BLOCK definition
- IDBUFFER: just a list of references to objects
- IMAGEDEF: IMAGE definition structure, required by the IMAGE entity
- IMAGEDEF_REACTOR: also required by the IMAGE entity
- LAYER_INDEX: container for LAYER names
- MLINESTYLE
- OBJECT_PTR
- RASTERVARIABLES
- SPATIAL_INDEX: is always written out empty to a DXF file. This object can be ignored.
- SPATIAL_FILTER
- SORTENTSTABLE: control for regeneration/redraw order of entities

- XRECORD: used to store and manage arbitrary data. This object is similar in concept to XDATA but is not limited by size or order. Not supported by R13c0 through R13c3.

Still missing the LAYOUT object, which is mandatory in DXF R2000 to manage multiple paperspace layouts. I don't know how DXF R13/R14 manages multiple layouts or if they even support this feature, but I don't care much about DXF R13/R14, because AutoCAD has no write support for this two formats anymore. Ezdxf tries to upgrade this two DXF versions to DXF R2000 with the advantage of only two different data models to support: DXF R12 and DXF R2000+

New objects introduced by DXF R2000:

- LAYOUT: management object for modelspace and multiple paperspace layouts
- ACDBPLACEHOLDER: surprise - just a place holder

New objects in DXF R2004:

- DIMASSOC
- LAYER_FILTER
- MATERIAL
- PLOTSETTINGS
- VBA_PROJECT

New objects in DXF R2007:

- DATATABLE
- FIELD
- LIGHTLIST
- RENDER
- RENDERENVIRONMENT
- MENTALRAYRENDERSETTINGS
- RENDERGLOBAL
- SECTION
- SUNSTUDY
- TABLESTYLE
- UNDERLAYDEFINITION
- VISUALSTYLE
- WIPEOUTVARIABLES

New objects in DXF R2013:

- GEODATA

New objects in DXF R2018:

- ACDBNAVISWORKSMODELDEF

Undocumented objects:

- SCALE
- ACDBSECTIONVIEWSTYLE
- FIELDLIST

Objects Organisation

Many objects in the OBJECTS section are organized in a tree-like structure of DICTIONARY objects.

Starting point for this data structure is the 'root' DICTIONARY with several entries to other DICTIONARY objects. The root DICTIONARY has to be the first object in the OBJECTS section. The management dicts for GROUP and LAYOUT objects are really important, but IMHO most of the other management tables are optional and for the most use cases not necessary. Ezdxf creates only these entries in the root dict and most of them pointing to an empty DICTIONARY:

- ACAD_COLOR: points to an empty DICTIONARY
- ACAD_GROUP: required
- ACAD_LAYOUT: required
- ACAD_MATERIAL: points to an empty DICTIONARY
- ACAD_MLEADERSTYLE: points to an empty DICTIONARY
- ACAD_MLINESTYLE: points to an empty DICTIONARY
- ACAD_PLOTSETTINGS: points to an empty DICTIONARY
- ACAD_PLOTSTYLENAME: required, points to ACDBDICTIONARYWDFLT with one entry: 'Normal'
- ACAD_SCALELIST: points to an empty DICTIONARY
- ACAD_TABLESTYLE: points to an empty DICTIONARY
- ACAD_VISUALSTYLE: points to an empty DICTIONARY

Root DICTIONARY content for DXF R2018

```
0
SECTION
2       <<< start of the OBJECTS section
OBJECTS
0       <<< root DICTIONARY has to be the first object in the OBJECTS section
DICTIONARY
5       <<< handle
C
330     <<< owner tag
0       <<< always #0, has no owner
100
AcDbDictionary
281     <<< hard owner flag
1
3       <<< first entry
ACAD_CIP_PREVIOUS_PRODUCT_INFO
350     <<< handle to target (pointer)
78B     <<< points to a XRECORD with product info about the creator application
3       <<< entry with unknown meaning, if I should guess: something with about␣
↪colors ...
ACAD_COLOR
350
4FB     <<< points to a DICTIONARY
3       <<< entry with unknown meaning
ACAD_DETAILVIEWSTYLE
350
7ED     <<< points to a DICTIONARY
```

(continues on next page)

(continued from previous page)

```
3       <<< GROUP management, mandatory in all DXF versions
ACAD_GROUP
350
4FC     <<< points to a DICTIONARY
3       <<< LAYOUT management, mandatory if more than the *active* paperspace is used
ACAD_LAYOUT
350
4FD     <<< points to a DICTIONARY
3       <<< MATERIAL management
ACAD_MATERIAL
350
4FE     <<< points to a DICTIONARY
3       <<< MLEADERSTYLE management
ACAD_MLEADERSTYLE
350
4FF     <<< points to a DICTIONARY
3       <<< MLINESTYLE management
ACAD_MLINESTYLE
350
500     <<< points to a DICTIONARY
3       <<< PLOTSETTINGS management
ACAD_PLOTSETTINGS
350
501     <<< points to a DICTIONARY
3       <<< plot style name management
ACAD_PLOTSTYLENAME
350
503     <<< points to a ACDBDICTIONARYWDFLT
3       <<< SCALE management
ACAD_SCALELIST
350
504     <<< points to a DICTIONARY
3       <<< entry with unknown meaning
ACAD_SECTIONVIEWSTYLE
350
7EB     <<< points to a DICTIONARY
3       <<< TABLESTYLE management
ACAD_TABLESTYLE
350
505     <<< points to a DICTIONARY
3       <<< VISUALSTYLE management
ACAD_VISUALSTYLE
350
506     <<< points to a DICTIONARY
3       <<< entry with unknown meaning
ACDB_RECOMPOSE_DATA
350
7F3
3       <<< entry with unknown meaning
AcDbVariableDictionary
350
7AE     <<< points to a DICTIONARY with handles to DICTIONARYVAR objects
0
DICTIONARY
...
...
0
```

(continues on next page)

(continued from previous page)

```
ENDSEC
```

6.13.2 DXF Structures

DXF Sections

HEADER Section

In DXF R12 and prior the HEADER section was optional, but since DXF R13 the HEADER section is mandatory. The overall structure is:

```
0           <<< Begin HEADER section
SECTION
2
HEADER
9
$ACADVER    <<< Header variable items go here
1
AC1009
...
0
ENDSEC      <<< End HEADER section
```

A header variable has a name defined by a `(9, Name)` tag and following value tags.

See also:

Documentation of *ezdxf* `HeaderSection` class.

DXF Reference: Header Variables

CLASSES Section

The CLASSES section contains CLASS definitions which are only important for Autodesk products, some DXF entities require a class definition or AutoCAD will not open the DXF file.

The CLASSES sections was introduced with DXF AC1015 (AutoCAD Release R13).

See also:

DXF Reference: About the DXF CLASSES Section

Documentation of *ezdxf* `ClassesSection` class.

The CLASSES section in DXF files holds the information for application-defined classes whose instances appear in the BLOCKS, ENTITIES, and OBJECTS sections of the database. It is assumed that a class definition is permanently fixed in the class hierarchy. All fields are required.

Update 2019-03-03:

Class names are not unique, Autodesk Architectural Desktop 2007 uses the same name, but with different CPP class names in the CLASS section, so storing classes in a dictionary by name as key caused loss of class entries in ezdxf, using a tuple of (name, cpp_class_name) as storage key solved the problem.

CLASS Entities

See also:

DXF Reference: Group Codes for the CLASS entity

CLASS entities have no handle and therefore ezdxf does not store the CLASS entity in the drawing entities database!

```
0
SECTION
2           <<< begin CLASSES section
CLASSES
0           <<< first CLASS entity
CLASS
1           <<< class DXF entity name; THIS ENTRY IS MAYBE NOT UNIQUE
ACDBDICTIONARYWDFLT
2           <<< C++ class name; always unique
AcDbDictionaryWithDefault
3           <<< application name
ObjectDBX Classes
90          <<< proxy capabilities flags
0
91          <<< instance counter for custom class, since DXF version AC1018 (R2004)
0           <<< no problem if the counter is wrong, AutoCAD doesn't care about
280         <<< was-a-proxy flag: 1= class was not loaded when this DXF file was␣
↪created
0           <<< 0= otherwise
281         <<< is-an-entity flag: 1= instances reside in the BLOCKS or ENTITIES␣
↪section
0           <<< 0= instances may appear only in the OBJECTS section
0           <<< next CLASS entity
CLASS
...
0           <<< end of CLASSES section
ENDSEC
```

TABLES Section

The TABLES section contains the resource tables of a DXF document.

APPID Table

The APPID table stores unique application identifiers. These identifiers are used to mark sub-sections in the XDATA section of DXF entities. AutoCAD will not load DXF files which uses AppIDs without an entry in the AppIDs table and the "ACAD" entry must always exist.

Some known AppIDs:

APPID	Used by	Description
ACAD	Autodesk	various use cases
AcAecLayerStandard	Autodesk	layer description
AcCmTransparency	Autodesk	layer transparency
HATCHBACKGROUNDCOLOR	Autodesk	background color for pattern fillings
EZDXF	ezdxf	meta data

See also:

- DXF Reference: TABLES Section
- DXF Reference: APPID Table
- *AppID* class

Table Structure DXF R12

```
0           <<< start of table
TABLE
2           <<< table type
APPID
70          <<< count of table entries, AutoCAD ignores this value
3
0           <<< 1. table entry
APPID
2           <<< unique application identifier
ACAD
70          <<< flags, see `APPID`_ reference
0           <<< in common cases always 0
0           <<< next table entry
APPID
...
0           <<< end of APPID table
ENDTAB
```

Table Structure DXF R2000+

```
0           <<< start of table
TABLE
2           <<< table type
APPID
5           <<< table handle
3
330         <<< owner tag, tables have no owner
0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of table entries, AutoCAD ignores this value
3
0           <<< first table entry
APPID
5           <<< handle of appid
2A
330         <<< owner handle, handle of APPID table
3
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbRegAppTableRecord
2           <<< unique application identifier
ACAD
70          <<< flags, see `APPID`_ reference
```

(continues on next page)

(continued from previous page)

```
0           <<< in common cases always 0
0           <<< next table entry
APPID
...
0           <<< end of APPID table
ENDTAB
```

Name References

APPID table entries are referenced by name:

- XDATA section of DXF entities

BLOCK_RECORD Table

Block records are essential elements for the entities management, each layout (modelspace and paperspace) and every block definition has a block record entry. This block record is the hard *owner* of the entities of layouts, each entity has an owner handle which points to a block record of the layout.

DIMSTYLE Table

The DIMSTYLE table stores all dimension style definitions of a DXF drawing.

You have access to the dimension styles table by the attribute `Drawing.dimstyles`.

See also:

- DXF Reference: TABLES Section
- DXF Reference: DIMSTYLE Table

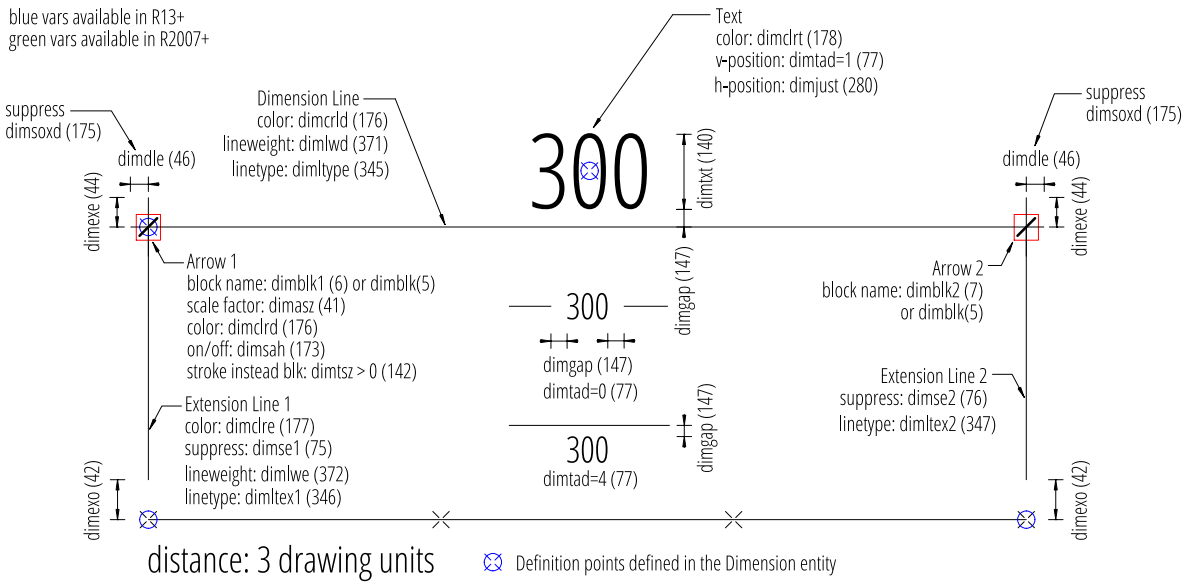
Table Structure DXF R12

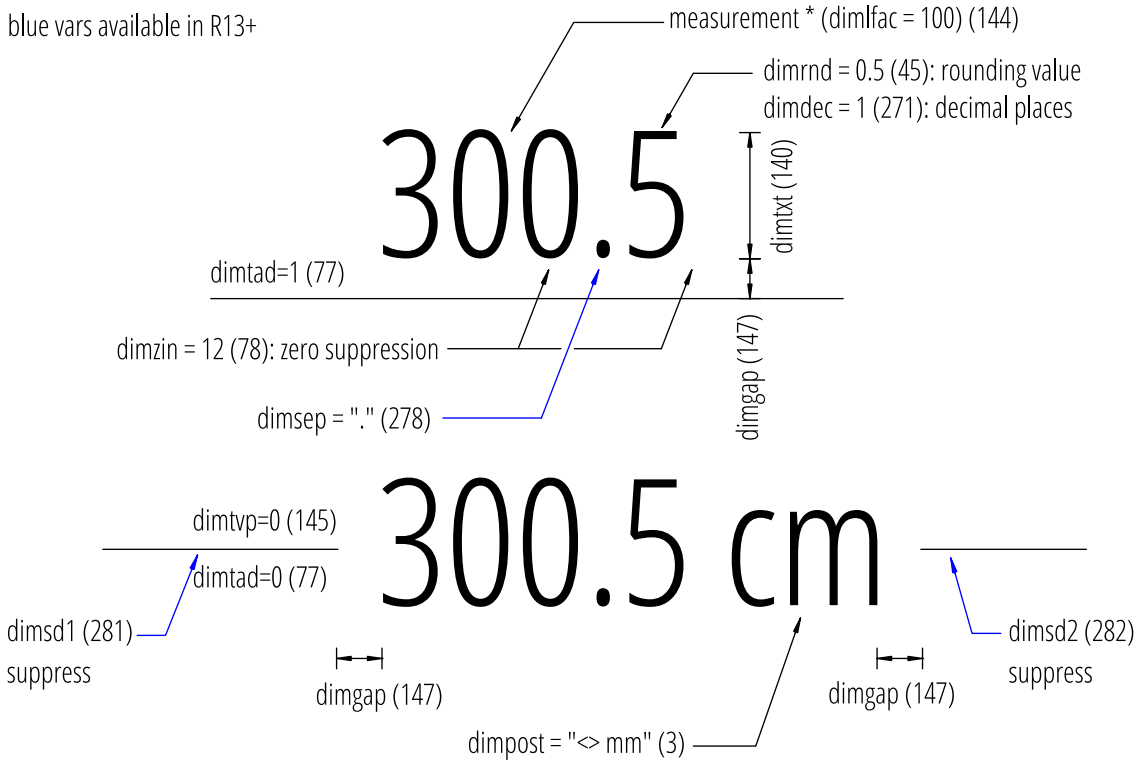
```
0           <<< start of table
TABLE
2           <<< set table type
DIMSTYLE
70          <<< count of line types defined in this table, AutoCAD ignores this value
9
0           <<< 1. DIMSTYLE table entry
DIMSTYLE
            <<< DIMSTYLE data tags
0           <<< 2. DIMSTYLE table entry
DIMSTYLE
            <<< DIMSTYLE data tags and so on
0           <<< end of DIMSTYLE table
ENDTAB
```

DIMSTYLE Entry DXF R12

DIMSTYLE Variables DXF R12

Source: CADDManager Blog





DIMVAR	Code	Description
DIMALT	170	Controls the display of alternate units in dimensions.
DIMALTD	171	Controls the number of decimal places in alternate units. If DIMALT is turned on, DIMALTD sets the number of digits displayed to the right of the decimal point in the alternate measurement.
DIMALTF	143	Controls the multiplier for alternate units. If DIMALT is turned on, DIMALTF multiplies linear dimensions by a factor to produce a value in an alternate system of measurement. The initial value represents the number of millimeters in an inch.
DIMAPOST	4	Specifies a text prefix or suffix (or both) to the alternate dimension measurement for all types of dimensions except angular. For instance, if the current units are Architectural, DIMALT is on, DIMALTF is 25.4 (the number of millimeters per inch), DIMALTD is 2, and DIMPOST is set to “mm”, a distance of 10 units would be displayed as 10”[254.00mm].
DIMASZ	41	Controls the size of dimension line and leader line arrowheads. Also controls the size of hook lines. Multiples of the arrowhead size determine whether dimension lines and text should fit between the extension lines. DIMASZ is also used to scale arrowhead blocks if set by DIMBLK. DIMASZ has no effect when DIMTSZ is other than zero.
DIMBLK	5	Sets the arrowhead block displayed at the ends of dimension lines.
DIMBLK1	6	Sets the arrowhead for the first end of the dimension line when DIMSAH is 1.
DIMBLK2	7	Sets the arrowhead for the second end of the dimension line when DIMSAH is 1.

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMCEN	141	Controls drawing of circle or arc center marks and centerlines by the DIMCENTER, DIMDIAMETER, and DIMRADIUS commands. For DIMDIAMETER and DIMRADIUS, the center mark is drawn only if you place the dimension line outside the circle or arc. • 0 = No center marks or lines are drawn • <0 = Centerlines are drawn • >0 = Center marks are drawn
DIMCLRD	176	Assigns colors to dimension lines, arrowheads, and dimension leader lines. • 0 = BYBLOCK • 1-255 = ACI AutoCAD Color Index • 256 = BYLAYER
DIMCLRE	177	Assigns colors to dimension extension lines, values like DIMCLRD
DIMCLRT	178	Assigns colors to dimension text, values like DIMCLRD
DIMDLE	46	Sets the distance the dimension line extends beyond the extension line when oblique strokes are drawn instead of arrowheads.
DIMDLI	43	Controls the spacing of the dimension lines in baseline dimensions. Each dimension line is offset from the previous one by this amount, if necessary, to avoid drawing over it. Changes made with DIMDLI are not applied to existing dimensions.
DIMEXE	44	Specifies how far to extend the extension line beyond the dimension line.
DIMEXO	42	Specifies how far extension lines are offset from origin points. With fixed-length extension lines, this value determines the minimum offset.

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMGAP	147	Sets the distance around the dimension text when the dimension line breaks to accommodate dimension text. Also sets the gap between annotation and a hook line created with the LEADER command. If you enter a negative value, DIMGAP places a box around the dimension text. DIMGAP is also used as the minimum length for pieces of the dimension line. When the default position for the dimension text is calculated, text is positioned inside the extension lines only if doing so breaks the dimension lines into two segments at least as long as DIMGAP. Text placed above or below the dimension line is moved inside only if there is room for the arrowheads, dimension text, and a margin between them at least as large as DIMGAP: 2 * (DIMASZ + DIMGAP).
DIMLFAC	144	Sets a scale factor for linear dimension measurements. All linear dimension distances, including radii, diameters, and coordinates, are multiplied by DIMLFAC before being converted to dimension text. Positive values of DIMLFAC are applied to dimensions in both modelspace and paperspace; negative values are applied to paperspace only. DIMLFAC applies primarily to nonassociative dimensions (DIMASSOC set 0 or 1). For nonassociative dimensions in paperspace, DIMLFAC must be set individually for each layout viewport to accommodate viewport scaling. DIMLFAC has no effect on angular dimensions, and is not applied to the values held in DIMRND, DIMTM, or DIMTP.

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMLIM	72	Generates dimension limits as the default text. Setting DIMLIM to On turns DIMTOL off. • 0 = Dimension limits are not generated as default text • 1 = Dimension limits are generated as default text
DIMPOST	3	Specifies a text prefix or suffix (or both) to the dimension measurement. For example, to establish a suffix for millimeters, set DIMPOST to mm; a distance of 19.2 units would be displayed as 19.2 mm. If tolerances are turned on, the suffix is applied to the tolerances as well as to the main dimension. Use "<>" to indicate placement of the text in relation to the dimension value. For example, enter "<>mm" to display a 5.0 millimeter radial dimension as "5.0mm". If you entered mm "<>", the dimension would be displayed as "mm 5.0".
DIMRND	45	Rounds all dimensioning distances to the specified value. For instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer. Note that the number of digits edited after the decimal point depends on the precision set by DIMDEC. DIMRND does not apply to angular dimensions.
DIMSAH	173	Controls the display of dimension line arrowhead blocks. • 0 = Use arrowhead blocks set by DIMBLK • 1 = Use arrowhead blocks set by DIMBLK1 and DIMBLK2

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMSCALE	40	Sets the overall scale factor applied to dimensioning variables that specify sizes, distances, or offsets. Also affects the leader objects with the LEADER command. Use MLEADERSCALE to scale multileader objects created with the MLEADER command. • 0.0 = A reasonable default value is computed based on the scaling between the current model space viewport and paperspace. If you are in paperspace or modelspace and not using the paperspace feature, the scale factor is 1.0. • >0 = A scale factor is computed that leads text sizes, arrowhead sizes, and other scaled distances to plot at their face values. DIMSCALE does not affect measured lengths, coordinates, or angles. Use DIMSCALE to control the overall scale of dimensions. However, if the current dimension style is annotative, DIMSCALE is automatically set to zero and the dimension scale is controlled by the CANNOSCALE system variable. DIMSCALE cannot be set to a non-zero value when using annotative dimensions.
DIMSE1	75	Suppresses display of the first extension line. • 0 = Extension line is not suppressed • 1 = Extension line is suppressed
DIMSE2	76	Suppresses display of the second extension line. • 0 = Extension line is not suppressed • 1 = Extension line is suppressed

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMSOXD	175	Suppresses arrowheads if not enough space is available inside the extension lines. • 0 = Arrowheads are not suppressed • 1 = Arrowheads are suppressed If not enough space is available inside the extension lines and DIMTIX is on, setting DIMSOXD to On suppresses the arrowheads. If DIMTIX is off, DIMSOXD has no effect.
DIMTAD	77	Controls the vertical position of text in relation to the dimension line. • 0 = Centers the dimension text between the extension lines. • 1 = Places the dimension text above the dimension line except when the dimension line is not horizontal and text inside the extension lines is forced horizontal (DIMTIH = 1). The distance from the dimension line to the baseline of the lowest line of text is the current DIMGAP value. • 2 = Places the dimension text on the side of the dimension line farthest away from the defining points. • 3 = Places the dimension text to conform to Japanese Industrial Standards (JIS). • 4 = Places the dimension text below the dimension line.
DIMTFAC	146	Specifies a scale factor for the text height of fractions and tolerance values relative to the dimension text height, as set by DIMTXT. For example, if DIMTFAC is set to 1.0, the text height of fractions and tolerances is the same height as the dimension text. If DIMTFAC is set to 0.7500, the text height of fractions and tolerances is three-quarters the size of dimension text.

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMTIH	73	Controls the position of dimension text inside the extension lines for all dimension types except Ordinate. • 0 = Aligns text with the dimension line • 1 = Draws text horizontally
DIMTIX	174	Draws text between extension lines. • 0 = Varies with the type of dimension. For linear and angular dimensions, text is placed inside the extension lines if there is sufficient room. For radius and diameter dimensions hat don't fit inside the circle or arc, DIMTIX has no effect and always forces the text outside the circle or arc. • 1 = Draws dimension text between the extension lines even if it would ordinarily be placed outside those lines
DIMTM	48	Sets the minimum (or lower) tolerance limit for dimension text when DIMTOL or DIMLIM is on. DIMTM accepts signed values. If DIMTOL is on and DIMTP and DIMTM are set to the same value, a tolerance value is drawn. If DIMTM and DIMTP values differ, the upper tolerance is drawn above the lower, and a plus sign is added to the DIMTP value if it is positive. For DIMTM, the program uses the negative of the value you enter (adding a minus sign if you specify a positive number and a plus sign if you specify a negative number).

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMTOFL	172	Controls whether a dimension line is drawn between the extension lines even when the text is placed outside. For radius and diameter dimensions (when DIMTIX is off), draws a dimension line inside the circle or arc and places the text, arrowheads, and leader outside. • 0 = Does not draw dimension lines between the measured points when arrowheads are placed outside the measured points • 1 = Draws dimension lines between the measured points even when arrowheads are placed outside the measured points
DIMTOH	74	Controls the position of dimension text outside the extension lines. • 0 = Aligns text with the dimension line • 1 = Draws text horizontally
DIMTOL	71	Appends tolerances to dimension text. Setting DIMTOL to on turns DIMLIM off.
DIMTP	47	Sets the maximum (or upper) tolerance limit for dimension text when DIMTOL or DIMLIM is on. DIMTP accepts signed values. If DIMTOL is on and DIMTP and DIMTM are set to the same value, a tolerance value is drawn. If DIMTM and DIMTP values differ, the upper tolerance is drawn above the lower and a plus sign is added to the DIMTP value if it is positive.
DIMTSZ	142	Specifies the size of oblique strokes drawn instead of arrowheads for linear, radius, and diameter dimensioning. • 0 = Draws arrowheads. • >0 = Draws oblique strokes instead of arrowheads. The size of the oblique strokes is determined by this value multiplied by the DIMSCALE value

continues on next page

Table 6 – continued from previous page

DIMVAR	Code	Description
DIMTVP	145	Controls the vertical position of dimension text above or below the dimension line. The DIMTVP value is used when DIMTAD = 0. The magnitude of the vertical offset of text is the product of the text height and DIMTVP. Setting DIMTVP to 1.0 is equivalent to setting DIMTAD = 1. The dimension line splits to accommodate the text only if the absolute value of DIMTVP is less than 0.7.
DIMTXT	140	Specifies the height of dimension text, unless the current text style has a fixed height.
DIMZIN	78	Controls the suppression of zeros in the primary unit value. Values 0-3 affect feet-and-inch dimensions only: • 0 = Suppresses zero feet and precisely zero inches • 1 = Includes zero feet and precisely zero inches • 2 = Includes zero feet and suppresses zero inches • 3 = Includes zero inches and suppresses zero feet • 4 (Bit 3) = Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000) • 8 (Bit 4) = Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5) • 12 (Bit 3+4) = Suppresses both leading and trailing zeros (for example, 0.5000 becomes .5)

Table Structure DXF R2000+

```
0           <<< start of table
TABLE
2           <<< set table type
DIMSTYLE
5           <<< DIMSTYLE table handle
5F
330         <<< owner tag, tables has no owner
0
100         <<< subclass marker
AcDbSymbolTable
```

(continues on next page)

(continued from previous page)

```
70          <<< count of dimension styles defined in this table, AutoCAD ignores this␣
↪value
9
0           <<< 1. DIMSTYLE table entry
DIMSTYLE
            <<< DIMSTYLE data tags
0           <<< 2. DIMSTYLE table entry
DIMSTYLE
            <<< DIMSTYLE data tags and so on
0           <<< end of DIMSTYLE table
ENDTAB
```

Additional DIMSTYLE Variables DXF R13/14

Source: CADDManager Blog

DIMVAR	code	Description
DIMADEC	179	Controls the number of precision places displayed in angular dimensions.
DIMALTTD	274	Sets the number of decimal places for the tolerance values in the alternate units of a dimension.
DIMALTTZ	286	Controls suppression of zeros in tolerance values.
DIMALTU	273	Sets the units format for alternate units of all dimension substyles except Angular.
DIMALTZ	285	Controls the suppression of zeros for alternate unit dimension values. DIMALTZ values 0-3 affect feet-and-inch dimensions only.
DIMAUNIT	275	Sets the units format for angular dimensions. • 0 = Decimal degrees • 1 = Degrees/minutes/seconds • 2 = Grad • 3 = Radians
DIMBLK_HANDLE	342	defines DIMBLK as handle to the BLOCK RECORD entry
DIMBLK1_HANDLE	343	defines DIMBLK1 as handle to the BLOCK RECORD entry
DIMBLK2_HANDLE	344	defines DIMBLK2 as handle to the BLOCK RECORD entry
DIMDEC	271	Sets the number of decimal places displayed for the primary units of a dimension. The precision is based on the units or angle format you have selected.
DIMDSEP	278	Specifies a single-character decimal separator to use when creating dimensions whose unit format is decimal. When prompted, enter a single character at the Command prompt. If dimension units is set to Decimal, the DIMDSEP character is used instead of the default decimal point. If DIMDSEP is set to NULL (default value, reset by entering a period), the decimal point is used as the dimension separator.
DIMJUST	280	Controls the horizontal positioning of dimension text. • 0 = Positions the text above the dimension line and center-justifies it between the extension lines • 1 = Positions the text next to the first extension line • 2 = Positions the text next to the second extension line • 3 = Positions the text above and aligned with the first extension line

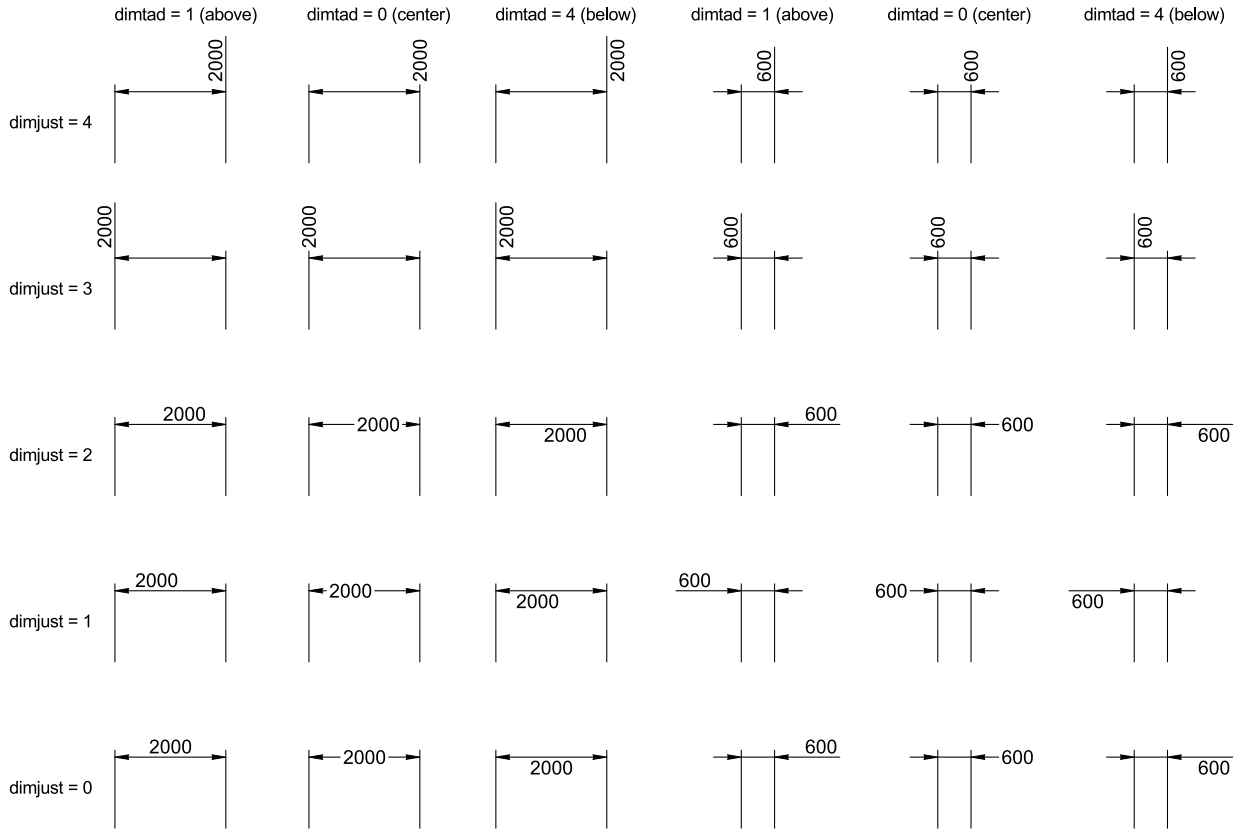
Additional DIMSTYLE Variables DXF R2000

Source: CADDManager Blog

DIMVAR	Code	Description
DIMALTRND	148	Rounds off the alternate dimension units.
DIMATFIT	289	Determines how dimension text and arrows are arranged when space is not sufficient to place both within the extension lines. • 0 = Places both text and arrows outside extension lines • 1 = Moves arrows first, then text • 2 = Moves text first, then arrows • 3 = Moves either text or arrows, whichever fits best A leader is added to moved dimension text when DIMTMOVE is set to 1.
DIMAZIN	79	Suppresses zeros for angular dimensions. • 0 = Displays all leading and trailing zeros • 1 = Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000) • 2 = Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5) • 3 = Suppresses leading and trailing zeros (for example, 0.5000 becomes .5)
DIMFRAC	276	Sets the fraction format when DIMLUNIT is set to 4 (Architectural) or 5 (Fractional). • 0 = Horizontal stacking • 1 = Diagonal stacking • 2 = Not stacked (for example, 1/2)
DIMLDRBLK_HANDLE	341	Specifies the arrow type for leaders. Handle to BLOCK RECORD
DIMLUNIT	277	Sets units for all dimension types except Angular. • 1 = Scientific • 2 = Decimal • 3 = Engineering • 4 = Architectural (always displayed stacked) • 5 = Fractional (always displayed stacked) • 6 = Microsoft Windows Desktop (decimal format using Control Panel settings for decimal separator and number grouping symbols)
DIMLWD	371	Assigns lineweight to dimension lines.

Text Location

This image shows the default text locations created by BricsCAD for dimension variables `dimtad` and `dimjust`:



Unofficial DIMSTYLE Variables for DXF R2007 and later

The following DIMVARS are **not documented** in the DXF Reference by Autodesk.

DIMVAR	Code	Description
DIMTFILL	69	Text fill 0=off; 1=background color; 2=custom color (see DIMTFILLCLR)
DIMTFILL-CLR	70	Text fill custom color as color index
DIMFXLON	290	Extension line has fixed length if set to 1
DIMFXL	49	Length of extension line below dimension line if fixed (DIMFXLON is 1), DIMEXE defines the the length above the dimension line
DIMJOGANG	50	Angle of oblique dimension line segment in jogged radius dimension
DIML-TYPE_HANDLE	345	Specifies the LINETYPE of the dimension line. Handle to LTYPE table entry
DIML-TEX1_HANDLE	346	Specifies the LINETYPE of the extension line 1. Handle to LTYPE table entry
DIML-TEX2_HANDLE	347	Specifies the LINETYPE of the extension line 2. Handle to LTYPE table entry

Extended Settings as Special XDATA Groups

Prior to DXF R2007, some extended settings for the dimension and the extension lines are stored in the XDATA section by following entries, this is not documented by Autodesk:

```
1001
ACAD_DSTYLE_DIM_LINETYPE        <<< linetype for dimension line
1070
380                             <<< group code, which differs from R2007 DIMDLTYPE
1005
FFFF                            <<< handle to LTYPE entry
1001
ACAD_DSTYLE_DIM_EXT1_LINETYPE   <<< linetype for extension line 1
1070
381                             <<< group code, which differs from R2007 DIMLTEX1
1005
FFFF                            <<< handle to LTYPE entry
1001
ACAD_DSTYLE_DIM_EXT2_LINETYPE   <<< linetype for extension line 1
1070
382                             <<< group code, which differs from R2007 DIMLTEX2
1005
FFFF                            <<< handle to LTYPE entry
1001
ACAD_DSTYLE_DIMEXT_ENABLED      <<< extension line fixed
1070
383                             <<< group code, which differs from R2007 DIMEXFIX
1070
1                               <<< fixed if 1 else 0
1001
ACAD_DSTYLE_DIMEXT_LENGTH       <<< extension line fixed length
1070
378                             <<< group code, which differs from R2007 DIMEXLEN
1040
1.33                            <<< length of extension line below dimension line
```

This XDATA groups requires also an appropriate APPID entry in the APPID table. This feature is not supported by *ezdxf*.

LAYER Table

TODO

See also:

- DXF Reference: TABLES Section
- DXF Reference: LAYER Table
- `Layer` class

Table Structure DXF R2000+

```
0           <<< start of table
TABLE
2           <<< name of table "LAYER"
LAYER
5           <<< handle of the TABLE
2
330         <<< owner tag is always "0"
0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of layers defined in this table, AutoCAD ignores this value
5
0           <<< 1. LAYER table entry
LAYER
...         <<< LAYER entity tags
0           <<< 2. LAYER table entry
LAYER
...         <<< LAYER entity tags
0           <<< end of TABLE
ENDTAB
```

Layer Entity Tags DXF R2000+

There are some quirks:

- the frozen/thawed state is stored in flags (group code 70)
- the locked/unlocked state is stored in flags (group code 70)
- the off state is stored as negative color value (group code 6)
- the layer description is stored in the XDATA section
- the transparency value is stored in the XDATA section

```
0           <<< LAYER table entry
LAYER
5           <<< handle of LAYER
10
330         <<< owner handle, handle of LAYER table
2
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLayerTableRecord
2           <<< layer name
0           <<< layer "0"
70          <<< flags
0
62          <<< color
7           <<< a negative value switches the layer off
420         <<< optional true color value
0
6           <<< linetype
Continuous
```

(continues on next page)

(continued from previous page)

```
290         <<< optional plot flag
1
370         <<< lineweight
-3
390         <<< handle to plot style
F
347         <<< material handle
47
348         <<< unknown1
0
1001        <<< XDATA section, APPID
AcAecLayerStandard
1000        <<< unknown first value, here an empty string

1000        <<< layer description
This layer has a description
1001        <<< APPID
AcCmTransparency
1071        <<< layer transparency value
0
```

Layer Viewport Overrides

Some layer attributes can be overridden individually for any VIEWPORT entity. This overrides are stored as extension dictionary entries of the LAYER entity pointing to XRECORD entities in the objects section:

```
0
LAYER
5
9F
102         <<< APP data, extension dictionary
{ACAD_XDICTIONARY
360         <<< handle to the xdict in the objects section
B3
102
}
330
2
100
AcDbSymbolTableRecord
100
AcDbLayerTableRecord
2
LayerA
...
```

The extension DICTIONARY entity:

```
0           <<< entity type
DICTIONARY
5           <<< handle
B3
330         <<< owner handle
9F          <<< the layer owns this dictionary
100         <<< subclass marker
```

(continues on next page)

(continued from previous page)

```
AcDbDictionary
280         <<< hard owned flag
1
281         <<< cloning type
1           <<< keep existing
3           <<< transparency override
ADSK_XREC_LAYER_ALPHA_OVR
360         <<< handle to XRECORD
E5
3           <<< color override
ADSK_XREC_LAYER_COLOR_OVR
360         <<< handle to XRECORD
B4
3           <<< linetype override
ADSK_XREC_LAYER_LINETYPE_OVR
360         <<< handle to XRECORD
DD
3           <<< lineweight override
ADSK_XREC_LAYER_LINEWT_OVR
360         <<< handle to XRECORD
E2
```

Transparency override XRECORD:

```
0           <<< entity type
XRECORD
5           <<< handle
E5
102         <<< reactors app data
{ACAD_REACTORS
330
B3          <<< extension dictionary
102
}
330         <<< owner tag
B3          <<< extension dictionary
100         <<< subclass marker
AcDbXrecord
280         <<< cloning flag
1           <<< keep existing
102         <<< for each overridden VIEWPORT one entry
{ADSK_LYR_ALPHA_OVERRIDE
335         <<< handle to VIEWPORT
AC
440         <<< transparency override
33554661
102
}
```

Color override XRECORD:

```
0
XRECORD
...         <<< like transparency XRECORD
102         <<< for each overridden VIEWPORT one entry
{ADSK_LYR_COLOR_OVERRIDE
335         <<< handle to VIEWPORT
```

(continues on next page)

(continued from previous page)

```
AF
420         <<< color override
-1023409925 <<< raw color value
102
}
```

Linetype override XRECORD:

```
0
XRECORD
...         <<< like transparency XRECORD
102         <<< for each overridden VIEWPORT one entry
{ADSK_LYR_LINETYPE_OVERRIDE
335         <<< handle to VIEWPORT
AC
343         <<< linetype override
DC          <<< handle to LINETYPE table entry
102
}
```

Lineweight override XRECORD:

```
0
XRECORD
...         <<< like transparency XRECORD
102         <<< for each overridden VIEWPORT one entry
{ADSK_LYR_LINEWT_OVERRIDE
335         <<< handle to VIEWPORT
AC
91          <<< lineweight override
13          <<< lineweight value
102
}
```

Name References

LAYER table entries are referenced by name:

- all graphical DXF entities
- VIEWPORT entity, frozen layers
- LAYER_FILTER
- LAYER_INDEX

LTYPE Table

The LTYPE table stores all line type definitions of a DXF drawing. Every line type used in the drawing has to have a table entry, or the DXF drawing is invalid for AutoCAD.

DXF R12 supports just simple line types, DXF R2000+ supports also complex line types with text or shapes included.

You have access to the line types table by the attribute `Drawing.linetypes`. The line type table itself is not stored in the entity database, but the table entries are stored in entity database, and can be accessed by its handle.

See also:

- DXF Reference: TABLES Section
- DXF Reference: LTYPE Table
- *Linetype* class

Table Structure DXF R12

```
0           <<< start of table
TABLE
2           <<< table type
LTYPE
70          <<< count of table entries, AutoCAD ignores this value
9
0           <<< 1. LTYPE table entry
LTYPE
            <<< LTYPE data tags
0           <<< 2. LTYPE table entry
LTYPE
            <<< LTYPE data tags and so on
0           <<< end of LTYPE table
ENDTAB
```

Table Structure DXF R2000+

```
0           <<< start of table
TABLE
2           <<< table type
LTYPE
5           <<< table handle
5F
330         <<< owner tag, tables have no owner
0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of table entiries, AutoCAD ignores this value
9
0           <<< 1. LTYPE table entry
LTYPE
            <<< LTYPE data tags
0           <<< 2. LTYPE table entry
LTYPE
            <<< LTYPE data tags and so on
```

(continues on next page)

(continued from previous page)

```
0           <<< end of LTYPE table
ENDTAB
```

Simple Line Type

ezdxf setup for line type "CENTER":

```
dwg.linetypes.add("CENTER",
    description="Center ____ _ ____ _ ____ _ ____ _ ____ _ ____",
    pattern=[2.0, 1.25, -0.25, 0.25, -0.25],
)
```

Simple Line Type Tag Structure DXF R2000+

```
0           <<< line type table entry
LTYPE
5           <<< handle of line type
1B1
330         <<< owner handle, handle of LTYPE table
5F
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLinetypeTableRecord
2           <<< line type name
CENTER
70          <<< flags
0
3
Center ____ _ ____ _ ____ _ ____ _ ____ _ ____
72          <<< signature tag
65          <<< ascii code for "A"
73          <<< count of pattern groups starting with a code 49 tag
4           <<< 4 pattern groups
40          <<< overall pattern length in drawing units
2.0
49          <<< 1. pattern group
1.25        <<< >0 line, <0 gap, =0 point
74          <<< type marker
0           <<< 0 for line group
49          <<< 2. pattern group
-0.25
74
0
49          <<< 3. pattern group
0.25
74
0
49          <<< 4. pattern group
-0.25
74
0
```

Complex Line Type TEXT

ezdxf setup for line type "GASLEITUNG":

```
dwg.linetypes.add("GASLEITUNG",
    description="Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--",
    length=1,
    pattern='A,.5,-.2,["GAS",STANDARD,S=.1,U=0.0,X=-0.1,Y=-.05],-.25',
)
```

TEXT Tag Structure

```
0
LTYPE
5
614
330
5F
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLinetypeTableRecord
2
GASLEITUNG
70
0
3
Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--
72          <<< signature tag
65          <<< ascii code for "A"
73          <<< count of pattern groups starting with a code 49 tag
3           <<< 3 pattern groups
40          <<< overall pattern length in drawing units
1
49          <<< 1. pattern group
0.5         <<< >0 line, <0 gap, =0 point
74          <<< type marker
0           <<< 0 for line group
49          <<< 2. pattern group
-0.2
74          <<< type marker
2           <<< 2 for text group
75          <<< shape number in shape-file
0           <<< always 0 for text group
340         <<< handle to text style "STANDARD"
11
46          <<< scaling factor: "s" in pattern definition
0.1
50          <<< rotation angle: "r" and "u" in pattern definition
0.0
44          <<< shift x units: "x" in pattern definition = parallel to line direction
-0.1
45          <<< shift y units: "y" in pattern definition = normal to line direction
-0.05
9           <<< text
```

(continues on next page)

(continued from previous page)

```
GAS
49          <<< 3. pattern group
-0.25
74
0
```

Complex Line Type SHAPE

ezdxf setup for line type 'GRENZE2':

```
dwg.linetypes.new('GRENZE2', dxfattribs={
    'description': 'Grenze eckig ----[]-----[]----[]-----[]----[]--',
    'length': 1.45,
    'pattern': 'A,.25,-.1,[132,ltypeshp.shx,x=-.1,s=.1],-.1,1',
})
```

SHAPE Tag Structure

```
0
LTYPE
5
615
330
5F
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLinetypeTableRecord
2
GRENZE2
70
0
3
Grenze eckig ----[]-----[]----[]-----[]----[]--
72          <<< signature tag
65          <<< ascii code for "A"
73          <<< count of pattern groups starting with a code 49 tag
4           <<< 4 pattern groups
40          <<< overall pattern length in drawing units
1.45
49          <<< 1. pattern group
0.25        <<< >0 line, <0 gap, =0 point
74          <<< type marker
0           <<< 0 for line group
49          <<< 2. pattern group
-0.1
74          <<< type marker
4           <<< 4 for shape group
75          <<< shape number in shape-file
132
340         <<< handle to shape-file entry "ltypeshp.shx"
616
46          <<< scaling factor: "s" in pattern definition
```

(continues on next page)

(continued from previous page)

```
0.1
50          <<< rotation angle: "r" and "u" in pattern definition
0.0
44          <<< shift x units: "x" in pattern definition = parallel to line direction
-0.1
45          <<< shift y units: "y" in pattern definition = normal to line direction
0.0
49          <<< 3. pattern group
-0.1
74
0
49          <<< 4. pattern group
1.0
74
0
```

Name References

LTYPE table entries are referenced by name:

- all graphical DXF entities
- LAYER table entry
- DIMSTYLE table entry
- DIMSTYLE override
- MLINESTYLE

STYLE Table

The STYLE table stores all text styles and shape-file definitions. The "STANDARD" entry must always exist.

Shape-files are also defined by a STYLE table entry, the bit 0 of the flags-tag is set to 1 and the name-tag is an empty string, the only important data is the font-tag with group code 3 which stores the associated SHX font file.

See also:

- DXF Reference: TABLES Section
- DXF Reference: STYLE Table
- *Textstyle* class

Table Structure DXF R12

```
0           <<< start of table
TABLE
2           <<< table type
STYLE
70           <<< count of table entries, AutoCAD ignores this value
1
0           <<< first table entry
STYLE
```

(continues on next page)

(continued from previous page)

```
2           <<< text style name
Standard
70          <<< flags, see `STYLE`_ reference
0
40          <<< fixed text height; 0 if not fixed
0.0
41          <<< width factor
1.0
50          <<< oblique angle
0.0
71          <<< text generation flags; 2=backwards (mirror-x), 4=upside down (mirror-
↪y)
0
42          <<< last height used
2.5
3           <<< font file name; SHX or TTF file name
txt
4           <<< big font name; SHX file with unicode symbols; empty if none

0           <<< next text style
STYLE
...
0           <<< end of STYLE table
ENDTAB
```

Table Structure DXF R2000+

```
0           <<< start of table
TABLE
2           <<< table type
STYLE
5           <<< table handle
5
330         <<< owner tag, tables have no owner
0
100         <<< subclass marker
AcDbSymbolTable
70           <<< count of table entries, AutoCAD ignores this value
1
0           <<< first table entry
STYLE
5           <<< handle of text style
29
330         <<< owner handle, handle of STYLE table
5
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbTextStyleTableRecord
2           <<< text style name
Standard
70          <<< flags, see `STYLE`_ reference
0
40          <<< fixed text height; 0 if not fixed
```

(continues on next page)

(continued from previous page)

```
0.0
41          <<< width factor
1.0
50          <<< oblique angle
0.0
71          <<< text generation flags; 2=backwards (mirror-x), 4=upside down (mirror-
↪y)
0
42          <<< last height used
2.5
3           <<< font file name; SHX or TTF file name
txt
4           <<< big font name; SHX file with unicode symbols; empty if none

0           <<< next text style
STYLE
...
0           <<< end of STYLE table
ENDTAB
```

Extended Font Data

Additional information of the font-family, italic and bold style flags are stored in the XDATA section of the STYLE entity by the APPID "ACAD":

```
0
STYLE
...
3
Arial.ttf
4

1001        <<< start of the XDATA section
ACAD        <<< APPID
1000        <<< font family name
Arial
1071        <<< style flags, see table below
50331682
```

Flag	dec	hex
ITALIC	16777216	0x1000000
BOLD	33554432	0x2000000

Name References

STYLE table entries are referenced by name:

- TEXT entity
- MTEXT entity
- DIMSTYLE table entry
- DIMSTYLE override

UCS Table

TODO

VIEW Table

The VIEW entry stores a named view of the model or a paperspace layout. This stored views makes parts of the drawing or some view points of the model in a CAD applications more accessible. This views have no influence to the drawing content or to the generated output by exporting PDFs or plotting on paper sheets, they are just for the convenience of CAD application users.

Using *ezdxf* you have access to the views table by the attribute `Drawing.views`. The views table itself is not stored in the entity database, but the table entries are stored in entity database, and can be accessed by its handle.

DXF R12

```
0
VIEW
2       <<< name of view
VIEWNAME
70      <<< flags bit-coded: 1st bit -> (0/1 = modelspace/paperspace)
0       <<< modelspace
40      <<< view width in Display Coordinate System (DCS)
20.01
10      <<< view center point in DCS
40.36   <<<     x value
20      <<<     group code for y value
15.86   <<<     y value
41      <<< view height in DCS
17.91
11      <<< view direction from target point, 3D vector
0.0     <<<     x value
21      <<<     group code for y value
0.0     <<<     y value
31      <<<     group code for z value
1.0     <<<     z value
12      <<< target point in WCS
0.0     <<<     x value
22      <<<     group code for y value
0.0     <<<     y value
32      <<<     group code for z value
0.0     <<<     z value
```

(continues on next page)

(continued from previous page)

```
42      <<< lens (focal) length
50.0    <<< 50mm
43      <<< front clipping plane, offset from target
0.0
44      <<< back clipping plane, offset from target
0.0
50      <<< twist angle
0.0
71      <<< view mode
0
```

See also:

Coordinate Systems

DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

```
0       <<< adding the VIEW table head, just for information
TABLE
2       <<< table name
VIEW
5       <<< handle of table, see owner tag of VIEW table entry
37C
330     <<< owner tag of table, always #0
0
100     <<< subclass marker
AcDbSymbolTable
70      <<< VIEW table (max.) count, not reliable (ignore)
9
0       <<< first VIEW table entry
VIEW
5       <<< handle
3EA
330     <<< owner, the VIEW table is the owner of the VIEW entry
37C     <<< handle of the VIEW table
100     <<< subclass marker
AcDbSymbolTableRecord
100     <<< subclass marker
AcDbViewTableRecord
2       <<< view name, from here all the same as DXF R12
VIEWNAME
70
0
40
20.01
10
40.36
20
15.86
41
17.91
11
0.0
21
```

(continues on next page)

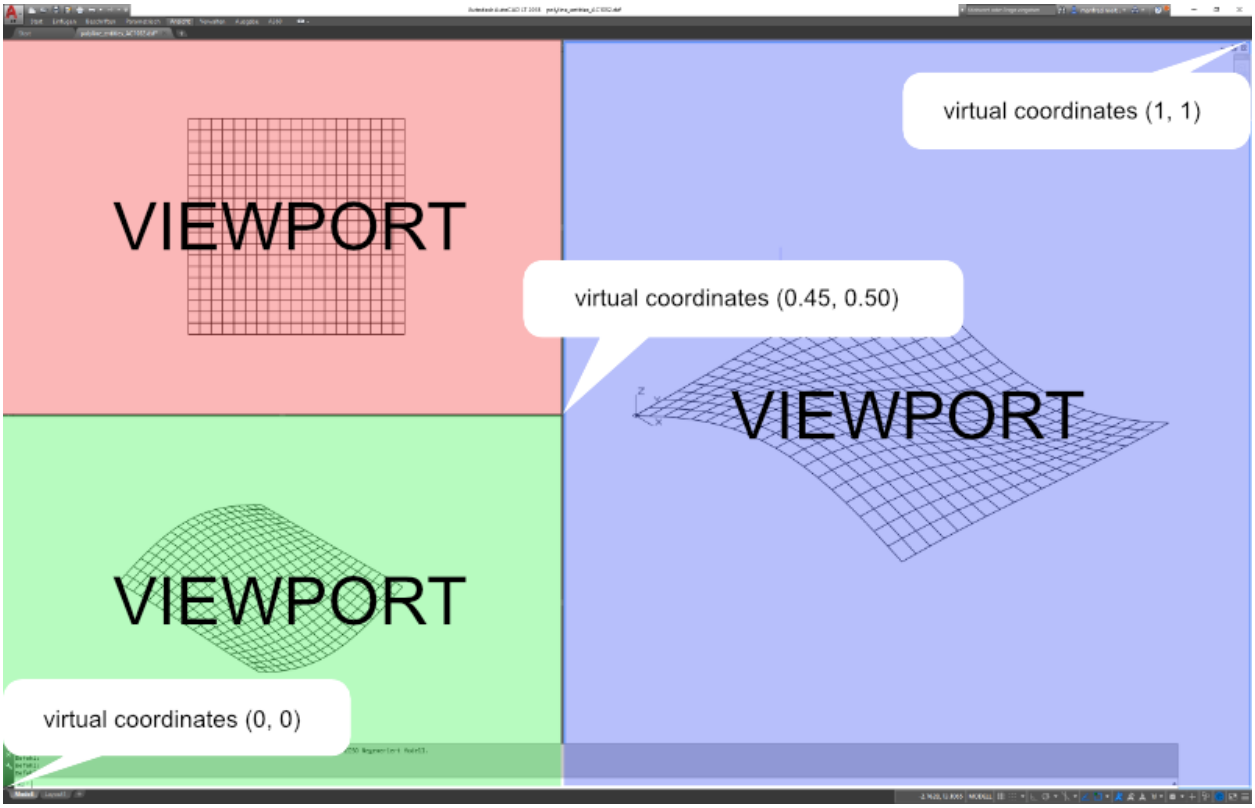
(continued from previous page)

```
0.0
31
1.0
12
0.0
22
0.0
32
0.0
42
50.0
43
0.0
44
0.0
50
0.0
71
0
281     <<< render mode 0-6 (... too much options)
0       <<< 0= 2D optimized (classic 2D)
72      <<< UCS associated (0/1 = no/yes)
0       <<< 0 = no
```

DXF R2000+ supports additional features in the VIEW entry, see the VIEW table reference provided by Autodesk.

VPORT Configuration Table

The VPORT table stores the modelspace viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport.



In contrast to other tables the VPORT table can have multiple entries with the same name, because all VPORT entries of a multi-viewport configuration are having the same name - the viewport configuration name. The name of the actual displayed viewport configuration is `'*ACTIVE'`, as always table entry names are case insensitive (`'*ACTIVE' == '*Active'`).

The available display area in AutoCAD has normalized coordinates, the lower-left corner is (0, 0) and the upper-right corner is (1, 1) regardless of the true aspect ratio and available display area in pixels. A single viewport configuration has one VPORT entry `'*ACTIVE'` with the lower-left corner (0, 0) and the upper-right corner (1, 1).

The following statements refer to a 2D plan view: the view-target-point defines the origin of the DCS (Display Coordinate system), the view-direction vector defines the z-axis of the *DCS*, the view-center-point (in DCS) defines the point in modelspace translated to the center point of the viewport, the view height and the aspect-ratio defines how much of the modelspace is displayed. AutoCAD tries to fit the modelspace area into the available viewport space e.g. view height is 15 units and aspect-ratio is 2.0 the modelspace to display is 30 units wide and 15 units high, if the viewport has an aspect ratio of 1.0, AutoCAD displays 30x30 units of the modelspace in the viewport. If the modelspace aspect-ratio is 1.0 the modelspace to display is 15x15 units and fits properly into the viewport area.

But tests show that the translation of the view-center-point to the middle of the viewport not always work as I expected. (still digging…)

Note: All floating point values are rounded to 2 decimal places for better readability.

DXF R12

Multi-viewport configuration with three viewports.

```
0       <<< table start
TABLE
2       <<< table type
VPORT
70      <<< VPORT table (max.) count, not reliable (ignore)
3
0       <<< first VPORT entry
VPORT
2       <<< VPORT (configuration) name
*ACTIVE
70      <<< standard flags, bit-coded
0
10      <<< lower-left corner of viewport
0.45    <<<     x value, virtual coordinates in range [0 - 1]
20      <<<     group code for y value
0.0     <<<     y value, virtual coordinates in range [0 - 1]
11      <<< upper-right corner of viewport
1.0     <<<     x value, virtual coordinates in range [0 - 1]
21      <<<     group code for y value
1.0     <<<     y value, virtual coordinates in range [0 - 1]
12      <<< view center point (in DCS), ???
13.71   <<<     x value
22      <<<     group code for y value
0.02    <<<     y value
13      <<< snap base point (in DCS)
0.0     <<<     x value
23      <<<     group code for y value
0.0     <<<     y value
14      <<< snap spacing X and Y
1.0     <<<     x value
24      <<<     group code for y value
1.0     <<<     y value
15      <<< grid spacing X and Y
0.0     <<<     x value
25      <<<     group code for y value
0.0     <<<     y value
16      <<< view direction from target point (in WCS), defines the z-axis of the DCS
1.0     <<<     x value
26      <<<     group code for y value
-1.0    <<<     y value
36      <<<     group code for z value
1.0     <<<     z value
17      <<< view target point (in WCS), defines the origin of the DCS
0.0     <<<     x value
27      <<<     group code for y value
0.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
40      <<< view height
35.22
41      <<< viewport aspect ratio
0.99
42      <<< lens (focal) length
```

(continues on next page)

(continued from previous page)

```
50.0    <<< 50mm
43      <<< front clipping planes, offsets from target point
0.0
44      <<< back clipping planes, offsets from target point
0.0
50      <<< snap rotation angle
0.0
51      <<< view twist angle
0.0
71      <<< view mode
0
72      <<< circle zoom percent
1000
73      <<< fast zoom setting
1
74      <<< UCSICON setting
3
75      <<< snap on/off
0
76      <<< grid on/off
0
77      <<< snap style
0
78      <<< snap isopair
0
0       <<< next VPORT entry
VPORT
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
22
9.41
...
...
0       <<< next VPORT entry
VPORT
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
```

(continues on next page)

(continued from previous page)

```
21
0.5
12
2.01
22
-9.33
...
...
0
ENDTAB
```

DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

```
0       <<< table start
TABLE
2       <<< table type
VPORT
5       <<< table handle
151F
330     <<< owner, table has no owner - always #0
0
100     <<< subclass marker
AcDbSymbolTable
70      <<< VPORT table (max.) count, not reliable (ignore)
3
0       <<< first VPORT entry
VPORT
5       <<< entry handle
158B
330     <<< owner, VPORT table is owner of VPORT entry
151F
100     <<< subclass marker
AcDbSymbolTableRecord
100     <<< subclass marker
AcDbViewportTableRecord
2       <<< VPORT (configuration) name
*ACTIVE
70      <<< standard flags, bit-coded
0
10      <<< lower-left corner of viewport
0.45    <<<     x value, virtual coordinates in range [0 - 1]
20      <<<     group code for y value
0.0     <<<     y value, virtual coordinates in range [0 - 1]
11      <<< upper-right corner of viewport
1.0     <<<     x value, virtual coordinates in range [0 - 1]
21      <<<     group code for y value
1.0     <<<     y value, virtual coordinates in range [0 - 1]
12      <<< view center point (in DCS)
13.71   <<<     x value
22      <<<     group code for y value
0.38    <<<     y value
13      <<< snap base point (in DCS)
0.0     <<<     x value
```

(continues on next page)

(continued from previous page)

```
23      <<<     group code for y value
0.0     <<<     y value
14      <<< snap spacing X and Y
1.0     <<<     x value
24      <<<     group code for y value
1.0     <<<     y value
15      <<< grid spacing X and Y
0.0     <<<     x value
25      <<<     group code for y value
0.0     <<<     y value
16      <<< view direction from target point (in WCS)
1.0     <<<     x value
26      <<<     group code for y value
-1.0    <<<     y value
36      <<<     group code for z value
1.0     <<<     z value
17      <<< view target point (in WCS)
0.0     <<<     x value
27      <<<     group code for y value
0.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
40      <<< view height
35.22
41      <<< viewport aspect ratio
0.99
42      <<< lens (focal) length
50.0    <<< 50mm
43      <<< front clipping planes, offsets from target point
0.0
44      <<< back clipping planes, offsets from target point
0.0
50      <<< snap rotation angle
0.0
51      <<< view twist angle
0.0
71      <<< view mode
0
72      <<< circle zoom percent
1000
73      <<< fast zoom setting
1
74      <<< UCSICON setting
3
75      <<< snap on/off
0
76      <<< grid on/off
0
77      <<< snap style
0
78      <<< snap isopair
0
281     <<< render mode 1-6 (... too many options)
0       <<< 0 = 2D optimized (classic 2D)
65      <<< Value of UCSVP for this viewport. (0 = UCS will not change when this␣
↪viewport is activated)
1       <<< 1 = then viewport stores its own UCS which will become the current UCS␣
```

(continues on next page)

(continued from previous page)

```
↪whenever the viewport is activated.
110     <<< UCS origin (3D point)
0.0     <<<     x value
120     <<<     group code for y value
0.0     <<<     y value
130     <<<     group code for z value
0.0     <<<     z value
111     <<< UCS X-axis (3D vector)
1.0     <<<     x value
121     <<<     group code for y value
0.0     <<<     y value
131     <<<     group code for z value
0.0     <<<     z value
112     <<< UCS Y-axis (3D vector)
0.0     <<<     x value
122     <<<     group code for y value
1.0     <<<     y value
132     <<<     group code for z value
0.0     <<<     z value
79      <<< Orthographic type of UCS 0-6 (... too many options)
0       <<< 0 = UCS is not orthographic
146     <<< elevation
0.0
1001    <<< extended data - undocumented
ACAD_NAV_VCDISPLAY
1070
3
0       <<< next VPORT entry
VPORT
5
158C
330
151F
100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
22
9.72
...
...
0       <<< next VPORT entry
VPORT
```

(continues on next page)

(continued from previous page)

```
5
158D
330
151F
100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
21
0.5
12
2.01
22
-8.97
...
...
0
ENDTAB
```

The TABLES section of DXF R13 and later looks like this:

```
0
SECTION
2           <<< begin TABLES section
TABLES
0           <<< first TABLE
TABLE
2           <<< name of table "LTYPE"
LTYPE
5           <<< handle of the TABLE
8
330         <<< owner handle is always "0"
0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of table entries
4           <<< do not rely on this value!
0           <<< first table entry
LTYPE
...
0           <<< second table entry
LTYPE
...
0           <<< end of TABLE
ENDTAB
0           <<< next TABLE
TABLE
```

(continues on next page)

(continued from previous page)

```
2           <<< name of table "LAYER"
LAYER
5           <<< handle of the TABLE
2
330         <<< owner handle is always "0"
0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of table entries
1
0           <<< first table entry
LAYER
...
0           <<< end of TABLE
ENDTAB
0           <<< end of SECTION
ENDSEC
```

The TABLES section of DXF R12 and prior is a bit simpler and does not contain the BLOCK_RECORD table. The handles in DXF R12 and prior are optional and only present if the HEADER variable $HANDLING is 1.

```
0
SECTION
2           <<< begin TABLES section
TABLES
0           <<< first TABLE
TABLE
2           <<< name of table "LTYPE"
LTYPE
5           <<< optional handle of the TABLE
8
70          <<< count of table entries
4
0           <<< first table entry
LTYPE
...
0           <<< second table entry
LTYPE
...
0           <<< end of TABLE
ENDTAB
0           <<< next TABLE
TABLE
2           <<< name of table "LAYER"
LAYER
5           <<< optional handle of the TABLE
2
70          <<< count of table entries
1
0           <<< first table entry
LAYER
...
0           <<< end of TABLE
ENDTAB
0           <<< end of SECTION
ENDSEC
```

BLOCKS Section

The BLOCKS section contains all BLOCK definitions, beside the *normal* reusable BLOCKS used by the INSERT entity, all layouts, as there are the modelspace and all paperspace layouts, have at least a corresponding BLOCK definition in the BLOCKS section. The name of the modelspace BLOCK is "*Model_Space" (DXF R12: "$MODEL_SPACE") and the name of the *active* paperspace BLOCK is "*Paper_Space" (DXF R12: "$PAPER_SPACE"), the entities of these two layouts are stored in the ENTITIES section, the *inactive* paperspace layouts are named by the scheme "*Paper_Spacennnn", and the content of the inactive paperspace layouts are stored in their BLOCK definition in the BLOCKS section.

The content entities of blocks are stored between the BLOCK and the ENDBLK entity.

BLOCKS section structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of BLOCKS section
BLOCKS
0           <<< start of 1. BLOCK definition
BLOCK
...         <<< Block content
...
0           <<< end of 1. Block definition
ENDBLK
0           <<< start of 2. BLOCK definition
BLOCK
...         <<< Block content
...
0           <<< end of 2. Block definition
ENDBLK
0           <<< end of BLOCKS section
ENDSEC
```

See also:

Block Management Structures Layout Management Structures

ENTITIES Section

TODO

OBJECTS Section

Objects in the OBJECTS section are organized in a hierarchical tree order, starting with the *named objects dictionary* as the first entity in the OBJECTS section (`Drawing.rootdict`).

Not all entities in the OBJECTS section are included in this tree, *Extension Dictionary* and XRECORD data of graphical entities are also stored in the OBJECTS section.

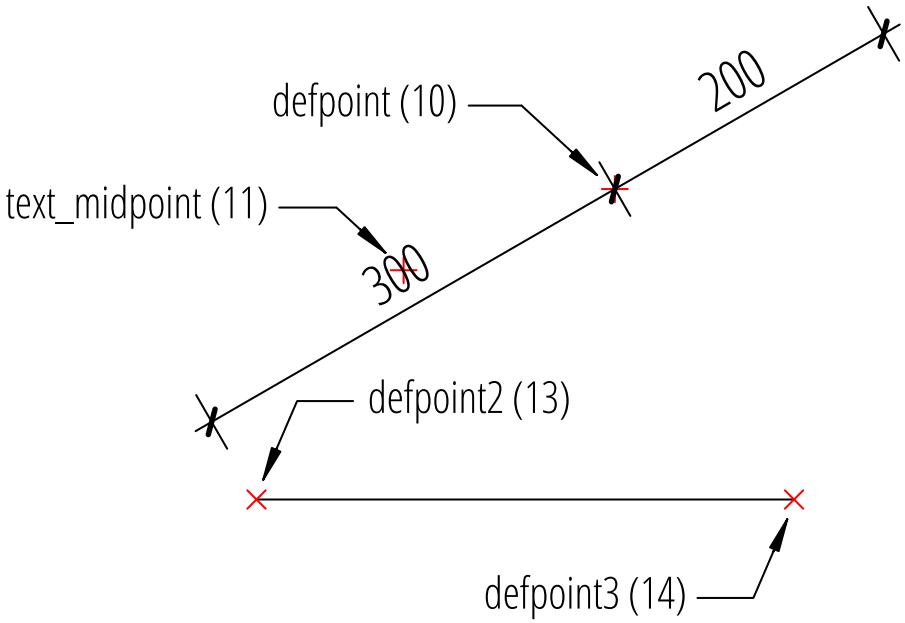
DXF Tables

DXF Entities

DIMENSION Internals

See also:

- DXF Reference: DIMENSION
- DXFInternals: *DIMSTYLE Table*



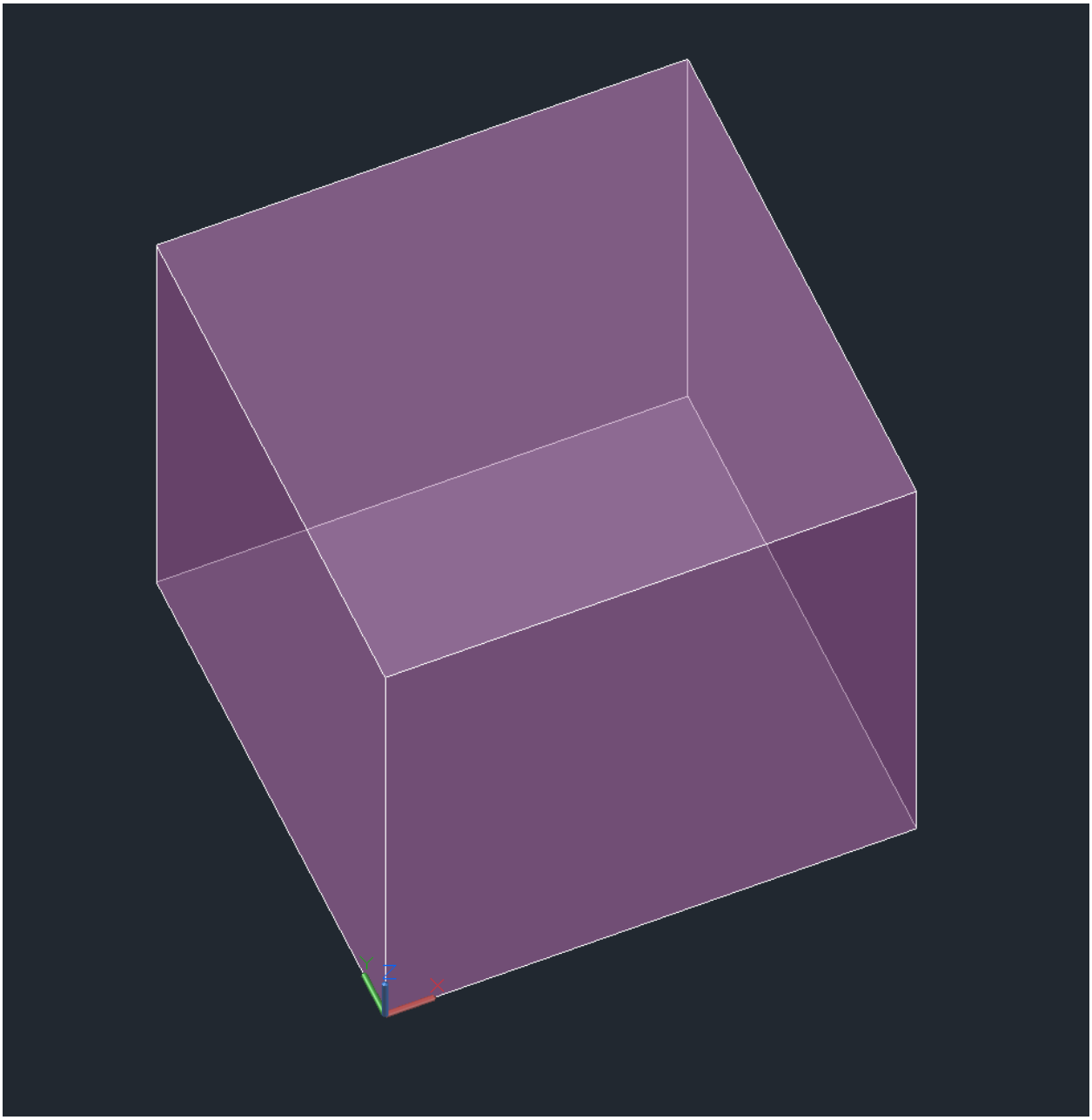
MESH Internals

The MESH entity is the compact version of the PolyFaceMesh implemented by the `Polyline` entity . The entity stores the vertices, edges and faces in a single entity and was introduced in DXF version R13/R14. For more information about the top level stuff go to the `Mesh` class.

See also:

- DXF Reference: MESH
- `ezdxf.entities.Mesh` class

The following DXF code represents this cube with subdivision level of 0:



```
0
MESH            <<< DXF type
5               <<< entity handle
2F
330             <<< block record handle of owner layout
17
100
AcDbEntity
8
0               <<< layer
62
6               <<< color
100
```

(continues on next page)

(continued from previous page)

```
AcDbSubDMesh    <<< subclass marker
71
2               <<< version
72
1               <<< blend crease, 1 is "on", 0 is "off"
91
0               <<< subdivision level is 0
92
8               <<< vertex count, 8 cube corners
10              <<< 1. vertex, x-axis
0.0
20              <<< y-axis
0.0
30              <<< z-axis
0.0
10              <<< 2. vertex
1.0
20
0.0
30
0.0
10              <<< 3. vertex
1.0
20
1.0
30
0.0
10              <<< 4. vertex
0.0
20
1.0
30
0.0
10              <<< 5. vertex
0.0
20
0.0
30
1.0
10              <<< 6. vertex
1.0
20
0.0
30
1.0
10              <<< 7. vertex
1.0
20
1.0
30
1.0
10              <<< 8. vertex
0.0
20
1.0
30
1.0
```

(continues on next page)

(continued from previous page)

```
93              <<< size of face list
30              <<< size  = count of group code 90 tags = 6 x 5
90              <<< vertex count of face 1
4               <<< MESH supports ngons, count = 3, 4, 5, 6 ...
90
0       <<< face 1, index of 1. vertex
90
3       <<< face 1, index of 2. vertex
90
2       <<< face 1, index of 3. vertex
90
1       <<< face 1, index of 4. vertex
90
4               <<< vertex count of face 2
90
4       <<< face 2, index of 1. vertex
90
5       <<< face 2, index of 2. vertex
90
6       <<< face 2, index of 3. vertex
90
7       <<< face 2, index of 4. vertex
90
4               <<< vertex count of face 3
90
0       <<< face 3, index of 1. vertex
90
1       <<< face 3, index of 2. vertex
90
5       <<< face 3, index of 3. vertex
90
4       <<< face 3, index of 4. vertex
90
4               <<< vertex count of face 4
90
1       <<< face 4, index of 1. vertex
90
2       <<< face 4, index of 2. vertex
90
6       <<< face 4, index of 3. vertex
90
5       <<< face 4, index of 4. vertex
90
4               <<< vertex count of face 5
90
3       <<< face 5, index of 1. vertex
90
7       <<< face 5, index of 2. vertex
90
6       <<< face 5, index of 3. vertex
90
2       <<< face 5, index of 4. vertex
90
4               <<< vertex count of face 6
90
0       <<< face 6, index of 1. vertex
90
```

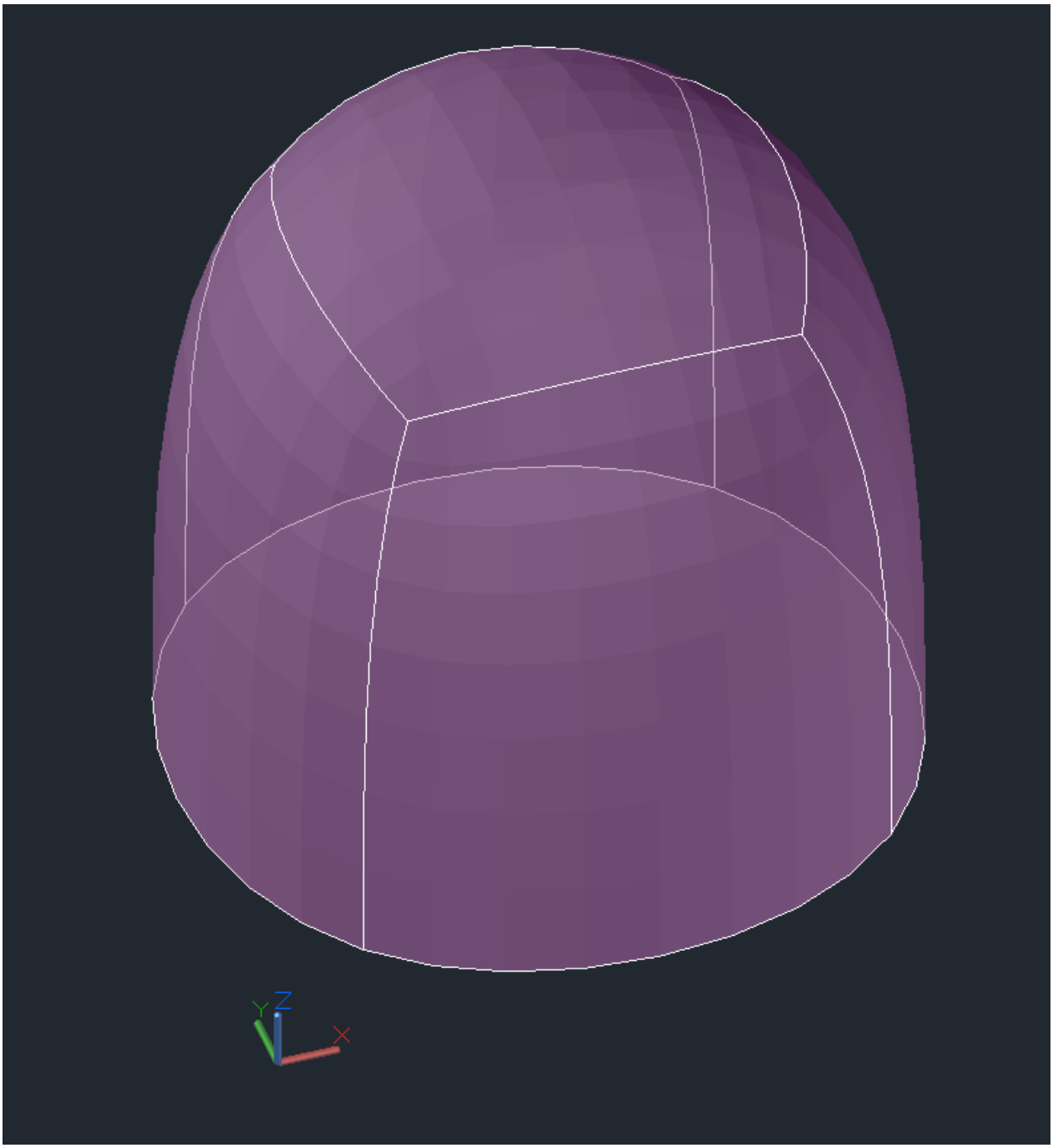
(continues on next page)

(continued from previous page)

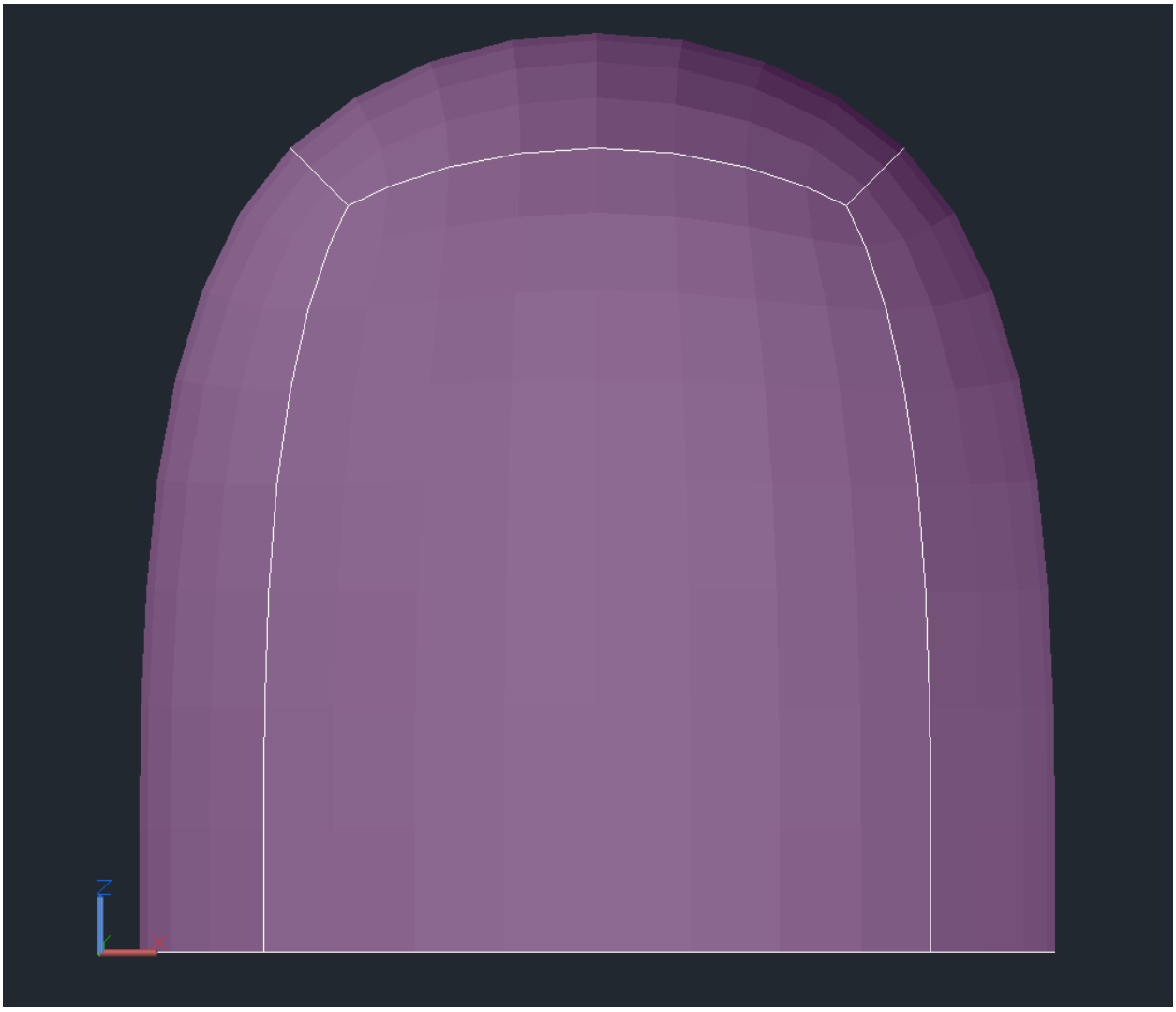
```
4       <<< face 6, index of 2. vertex
90
7       <<< face 6, index of 3. vertex
90
3       <<< face 6, index of 4. vertex
94              <<< edge count, each edge has exact two group code 90 tags
4               <<< the real edge count not the group code 90 tags!
90
0       <<< edge 1, vertex 1
90
1       <<< edge 1, vertex 1
90
1       <<< edge 2, vertex 1
90
2       <<< edge 2, vertex 2
90
2       <<< edge 3, vertex 1
90
3       <<< edge 3, vertex 2
90
3       <<< edge 4, vertex 1
90
0       <<< edge 4, vertex 2
95              <<< edge crease count, has to match edge count!
4
140
3.0     <<< crease value for edge 1
140
3.0     <<< crease value for edge 2
140
3.0     <<< crease value for edge 3
140
3.0     <<< crease value for edge 4
90              <<< property overwrite???
0
```

The edge and crease data have only a meaning if subdivision of the geometry will be applied! A crease value equal to the subdivision level prevents subdividing for the edge completely, a value between 0.0 and the subdivision level applies subdivision partially.

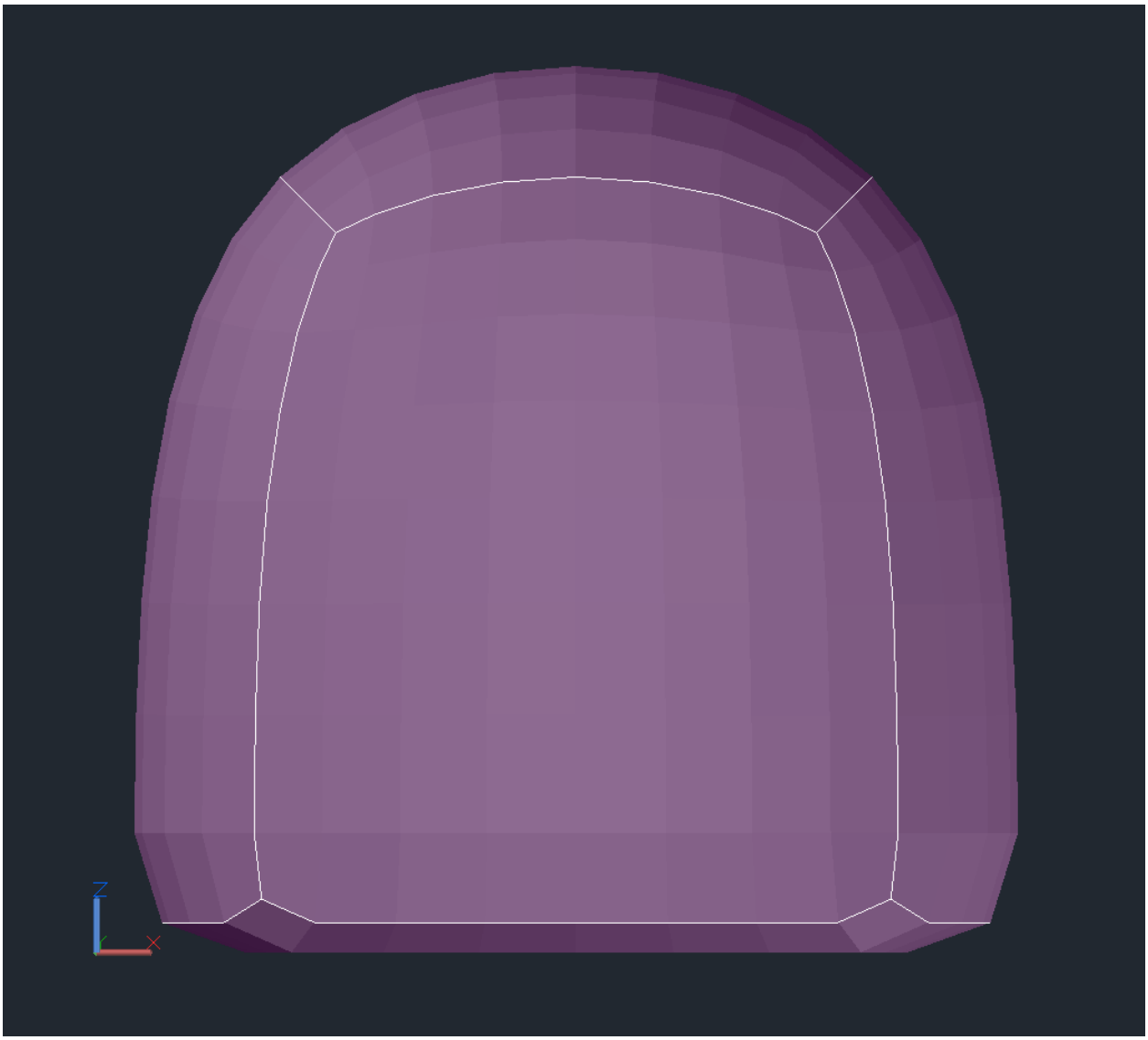
The cube with subdivision level of 3 and crease values of 3.0:



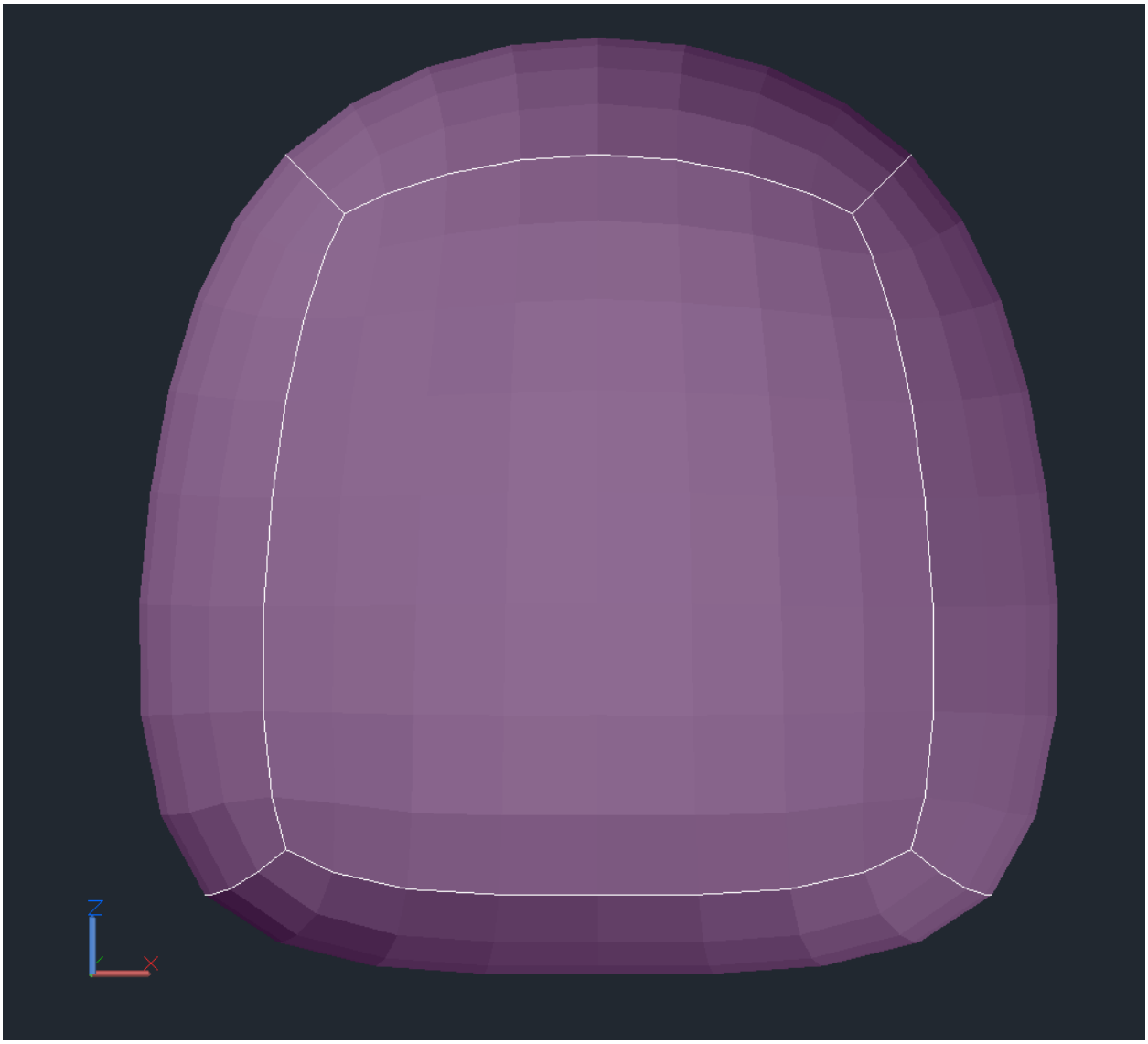
Front view for better details:



The cube with subdivision levels of 3 and crease values of 2.0:



The cube with subdivision level of 3 and crease values of 1.0:



The property overriding protocol is not documented in the DXF reference and currently I have no access to a CAD application which can created property overriding.

MULTILEADER Internals

The MULTILEADER leader is a very complex entity and has also some weird and unique properties.

1. MULTILEADER has the alias name MLEADER which is accepted by any *reliable CAD application*, but all of them create the entity as MULTILEADER
2. uses *raw-color* values to define colors
3. creates a complex context data structures beyond simple tags inside the subclass `AcDbMLeader`

See also:

- *ezdxf.entities.MultiLeader*
- *ezdxf.entities.MLeaderStyle*

- *ezdxf.render.MultiLeaderBuilder*
- *Tutorial for MultiLeader*
- DXF Reference: MLEADER

Example for *ezdxf.entities.MLeaderContext* created by BricsCAD:

```
300 <str> CONTEXT_DATA{
40 <float> 1.0    <<< content scale
10 <point> (x, y, z)      <<< content base point
41 <float> 4.0    <<< text height
140 <float> 4.0   <<< arrowhead size
145 <float> 2.0   <<< landing gap size
174 <int> 1       <<< doc missing
175 <int> 1       <<< doc missing
176 <int> 0       <<< doc missing
177 <int> 0       <<< doc missing
290 <int> 1       <<< has_mtext_content
<<< START MText Content tags:
304 <str> MTEXT content string
11 <point> (0.0, 0.0, 1.0)    <<< extrusion vector
340 <hex> #A0                 <<< text style as handle
12 <point> (x, y, z)          <<< text location
13 <point> (1.0, 0.0, 0.0)    <<< text direction
42 <float> 0.0        <<< text rotation
43 <float> 0.0        <<< text width
44 <float> 0.0        <<< text height
45 <float> 1.0        <<< text line space factor
170 <int> 1           <<< text line space style
90 <int> -1056964608  <<< text color (raw value)
171 <int> 1           <<< text attachment
172 <int> 1           <<< text flow direction
91 <int> -939524096   <<< text background color (raw value)
141 <float> 1.5       <<< text background scale factor
92 <int> 0            <<< text background transparency
291 <int> 0           <<< has_text_bg_color
292 <int> 0           <<< has_text_bg_fill
173 <int> 0           <<< text column type
293 <int> 0           <<< use text auto height
142 <float> 0.0       <<< text column width
143 <float> 0.0       <<< text column gutter width
294 <int> 0           <<< text column flow reversed
144 <float> missing   <<< text column height (optional?)
295 <int> 0           <<< text use word break
<<< END MText Content tags:
296 <int> 0       <<< has_block_content
<<< START Block content tags
<<< END Block content tags
110 <point> (0.0, 0.0, 0.0)       <<< MLEADER plane origin point
111 <point> (1.0, 0.0, 0.0)       <<< MLEADER plane x-axis direction
112 <point> (0.0, 1.0, 0.0)       <<< MLEADER plane y-axis direction
297 <int> 0                       <<< MLEADER normal reversed
302 <str> LEADER{
...
303 <str> }
302 <str> LEADER{
...
303 <str> }
```

(continues on next page)

(continued from previous page)

```
272 <int> 9       <<< doc missing
273 <int> 9       <<< doc missing
301 <str> }
<<< BricsCAD example for block content:
300 <str> CONTEXT_DATA{
40 <float> 1.0
10 <point> (x, y, z)
41 <float> 4.0
140 <float> 4.0
145 <float> 2.0
174 <int> 1
175 <int> 1
176 <int> 0
177 <int> 0
290 <int> 0       <<< has_mtext_content
296 <int> 1       <<< has_block_content
<<< START Block content tags
341 <hex> #94                 <<< dxf.block_record_handle
14 <point> (0.0, 0.0, 1.0)    <<< Block extrusion vector
15 <point> (x, y, z)          <<< Block location
16 <point> (1.0, 1.0, 1.0)    <<< Block scale vector, the x-, y- and z-axis scaling␣
↪factors
46 <float> 0.0                <<< Block rotation in radians!
93 <int> -1056964608          <<< Block color (raw value)
47 <float> 1.0                <<< start of transformation matrix (16x47)
47 <float> 0.0
47 <float> 0.0
47 <float> 18.427396871473
47 <float> 0.0
47 <float> 1.0
47 <float> 0.0
47 <float> 0.702618780008
47 <float> 0.0
47 <float> 0.0
47 <float> 1.0
47 <float> 0.0
47 <float> 0.0
47 <float> 0.0
47 <float> 0.0
47 <float> 1.0                <<< end of transformation matrix
<<< END Block content tags
110 <point> (0.0, 0.0, 0.0)       <<< MLEADER plane origin point
111 <point> (1.0, 0.0, 0.0)       <<< MLEADER plane x-axis direction
112 <point> (0.0, 1.0, 0.0)       <<< MLEADER plane y-axis direction
297 <int> 0                       <<< MLEADER normal reversed
302 <str> LEADER{
...
303 <str> }
272 <int> 9
273 <int> 9
301 <str> }
<<< Attribute content and other redundant block data is stored in the AcDbMLeader
<<< subclass:
100 <ctrl> AcDbMLeader
270 <int> 2                   <<< dxf.version
300 <str> CONTEXT_DATA{       <<< start context data
...
```

(continues on next page)

(continued from previous page)

```
301 <str> }                    <<< end context data
340 <hex> #6D                  <<< dxf.style_handle
90 <int> 6816768               <<< dxf.property_override_flags
...                            <<< property overrides
292 <int> 0                    <<< dxf.has_frame_text
<<< mostly redundant block data:
344 <hex> #94                  <<< dxf.block_record_handle
93 <int> -1056964608           <<< dxf.block_color (raw value)
10 <point> (1.0, 1.0, 1.0)     <<< dxf.block_scale_vector
43 <float> 0.0                 <<< dxf.block_rotation in radians!
176 <int> 0                    <<< dxf.block_connection_type
293 <int> 0                    <<< dxf.is_annotative
<<< REPEAT: (optional)
94 <int>                       <<< arrow head index?
345 <hex>                      <<< arrow head handle
<<< REPEAT: (optional)
330 <hex> #A3                  <<< ATTDEF handle
177 <int> 1                    <<< ATTDEF index
44 <float> 0.0                 <<< ATTDEF width
302 <str> B                    <<< ATTDEF text (reused group code)
...  common group codes 294, 178, 179, ...
```

MTEXT Internals

The MTEXT entity stores multiline text in a single entity and was introduced in DXF version R13/R14. For more information about the top level stuff go to the *MText* class.

See also:

- DXF Reference: MTEXT
- *ezdxf.entities.MText* class

Orientation

The MTEXT entity does not establish an OCS. The entity has a `text_direction` attribute, which defines the local x-axis, the `extrusion` attribute defines the normal vector and the y-axis = extrusion cross x-axis.

The MTEXT entity can have also a `rotation` attribute (in degrees), the x-axis attribute has higher priority than the `rotation` attribute, but it is not clear how to convert the `rotation` attribute into a `text_direction` vector, but for most common cases, where only the `rotation` attribute is present, the `extrusion` is most likely the WCS z-axis and the `rotation` is the direction in the xy-plane.

Text Content

The content text is divided across multiple tags of group code 3 and 1, the last line has the group code 1, each line can have a maximum line length of 255 bytes, but BricsCAD (and AutoCAD?) store only 249 bytes in single line and one byte is not always one char.

Inline Code Specials

The text formatting is done by inline codes, see the *MText* class.

Information gathered by implementing the `MTextEditor` and the `MTextParser` classes:

- **caret encoded characters:**
  - "^I" tabulator
  - "^J" (LF) is a valid line break like "\P"
  - "^M" (CR) is ignored
  - other characters render as empty square "□"
  - a space " " after the caret renders the caret glyph: "1^ 2" renders "1^2"
- **special encoded characters:**
  - "%%c" and "%%C" renders "Ø" (alt-0216)
  - "%%d" and "%%D" renders "°" (alt-0176)
  - "%%p" and "%%P" renders "±" (alt-0177)
- **Alignment command "\A": argument "0", "1" or "2" is expected**
  - the terminator symbol ";" is optional
  - the arguments "3", "4", "5", "6", "7", "8", "9" and "-" default to 0
  - other characters terminate the command and will be printed: "\AX", renders "X"
- **ACI color command "\C": int argument is expected**
  - the terminator symbol ";" is optional
  - a leading "-" or "+" terminates the command, "\C+5" renders "\C+5"
  - arguments > 255, are ignored but consumed "\C1000" renders nothing, not even a "0"
  - a trailing ";" after integers is always consumed, even for much to big values, "\C10000;" renders nothing
- **RGB color command "\c": int argument is expected**
  - the terminator symbol ";" is optional
  - a leading "-" or "+" terminates the command, "\c+255" renders "\c+255"
  - arguments >= 16777216 are masked by: value & 0xFFFFFF
  - a trailing ";" after integers is always consumed, even for much to big values, "\c9999999999;" renders nothing and switches the color to yellow (255, 227, 11)
- **Height command "\H" and "\H…x": float argument is expected**
  - the terminator symbol ";" is optional
  - a leading "-" is valid, but negative values are ignored
  - a leading "+" is valid
  - a leading "." is valid like "\H.5x" for height factor 0.5
  - exponential format is valid like "\H1e2" for height factor 100 and "\H1e-2" for 0.01
  - an invalid floating point value terminates the command, "\H1..5" renders "\H1..5"
- **Other commands with floating point arguments like the height command:**

  - Width commands “\W” and “\W…x”
  - Character tracking commands “\T” and “\T…x”, negative values are used
  - Slanting (oblique) command “\Q”

- **Stacking command “\S”:**
  - build fractions: “numerator (upr)” + “stacking type char (t)” + “denominator (lwr)” + “;”
  - divider chars: “^”, “/” or “#”
  - a space “ “ after the divider char “^” is mandatory to avoid caret decoding: “\S1^ 2;”
  - the terminator symbol “;” is mandatory to end the command, all chars beyond the “\S” until the next “;” or the end of the string are part of the fraction
  - backslash escape “\;” to render the terminator char
  - a space “ “ after the divider chars “/” and “#” is rendered as space “ ” in front of the denominator
  - the numerator and denominator can contain spaces
  - backslashes “\” inside the stacking command are ignored (except “\;”) “\S\N^ \P” render “N” over “P”, therefore property changes (color, text height, …) are not possible inside the stacking command
  - grouping chars “{” and “}” render as simple curly braces
  - caret encoded chars are decoded “^I”, “^J”, “^M”, but render as a simple space “ “ or as the replacement char “ ” plus a space
  - a divider char after the first divider char, renders as the char itself: “\S1/2/3” renders the horizontal fraction “1” / “2/3”

- **Font command “\f” and “\F”: export only “\f”, parse both, “\F” ignores some arguments**
  - the terminator symbol “;” is mandatory to end the command, all chars beyond the “\f” until the next “;” or the end of the string are part of the command
  - the command arguments are separated by the pipe char “|”
  - arguments: “font family name” | “bold” | “italic” | “codepage” | “pitch”; example “\fArial|b0|i0|c0|p0;”
  - only the “font family name” argument is required, fonts which are not available on the system are replaced by the “TXT.SHX” shape font
  - the “font family name” is the font name shown in font selection widgets in desktop applications
  - “b1” to use the bold font style, any other second char is interpreted as “non bold”
  - “i1” to use an italic font style, any other second char is interpreted as “non italic”
  - “c???” change codepage, “c0” use the default codepage, because of the age of unicode no further investigations, also seems to be ignored by AutoCAD and BricsCAD
  - “p???” change pitch size, “p0” means don’t change, ignored by AutoCAD and BricsCAD, to change the text height use the “\H” command
  - the order is not important, but export always in the shown order: “\fArial|b0|i0;” the arguments “c0” and “p0” are not required

- **Paragraph properties command “\p”**
  - the terminator symbol “;” is mandatory to end the command, all chars beyond the “\p” until the next “;” or the end of the string are part of the command
  - the command arguments are separated by commas “,”

- all values are factors for the initial char height of the MTEXT entity, example: char height = 2.5, "\pl1;" set the left paragraph indentation to 1 x 2.5 = 2.5 drawing units.
- all values are floating point values, see height command
- arguments are "i", "l", "r", "q", "t"
- a "*" as argument value, resets the argument to the initial value: "i0", "l0", "r0", the "q" argument most likely depends on the text direction; I haven't seen "t*". The sequence used by BricsCAD to reset all values is `"\pi*,l*,r*,q*,t;"`
- "i" indentation of the first line relative to the "l" argument as floating point value, "\pi1.5"
- "l" left paragraph indentation as floating point value, "\pl1.5"
- "r" right paragraph indentation as floating point value, "\pr1.5"
- "x" is required if a "q" or a "t" argument is present, the placement of the "x" has no obvious rules
- "q" paragraph alignment
  - "ql" left paragraph alignment
  - "qr" right paragraph alignment
  - "qc" center paragraph alignment
  - "qj" justified paragraph alignment
  - "qd" distributed paragraph alignment
- "t" tabulator stops as comma separated list, the default tabulator stops are located at 4, 8, 12, …, by defining at least one tabulator stop, the default tabulator stops will be ignored. There 3 kind of tabulator stops: left, right and center adjusted stops, e.g. "pxt1,r5,c8":
  - a left adjusted stop has no leading char, two left adjusted stops "\pxt1,2;"
  - a right adjusted stop has a preceding "r" char, "\pxtr1,r2;"
  - a center adjusted stop has a preceding "c" char, "\pxtc1,c2;"

  complex example to create a numbered list with two items: `"pxi-3,l4t4;1.^Ifirst item\P2.^Isecond item"`
- a parser should be very flexible, I have seen several different orders of the arguments and placing the sometimes required "x" has no obvious rules.
- exporting seems to be safe to follow these three rules:
  1. the command starts with "\px", the "x" does no harm, if not required
  2. argument order "i", "l", "r", "q", "t", any of the arguments can be left off
  3. terminate the command with a ";"

Height Calculation

There is no reliable way to calculate the MTEXT height from the existing DXF attributes. The `rect_height` (group code 43) attribute is not required and seldom present. DXF R2007 introduced the `defined_height` attribute to store the defined column height of the MTEXT entity but only in column mode. MTEXT entities without columns, except MTEXT entities created with column type "No Columns", store always 0.0 as defined column height. Which seems to mean: defined by the rendered text content.

The only way to calculate the MTEXT height is to replicate the rendering results of AutoCAD/BricsCAD by implementing a rendering engine for MTEXT.

In column mode the MTEXT height is stored for every column for DXF version before R2018. In DXF R2018+ the column heights are only stored if `MTextColumns.auto_height` is `False`. If `MTextColumns.auto_height` is `True`. But DXF R2018+ stores the MTEXT total width and height in explicit attributes.

Width Calculation

The situation for width calculation is better than for the height calculation, but the attributes `width` and `rect_width` are not mandatory.

There is a difference between MTEXT entities with and without columns:

Without columns the attribute `width` (reference column width) contains the true entity width if present. A long word can overshoot this width! The `rect_width` attribute is seldom present.

For MTEXT with columns, the `width` attribute is maybe wrong, the correct width for a column is stored in the `column_width` attribute and the `total_width` attribute stores the total width of the MTEXT entity overall columns, see also following section "Column Support".

Background Filling

The background fill support is available for DXF R2007+. The group code 90 defines the kind of background fill:

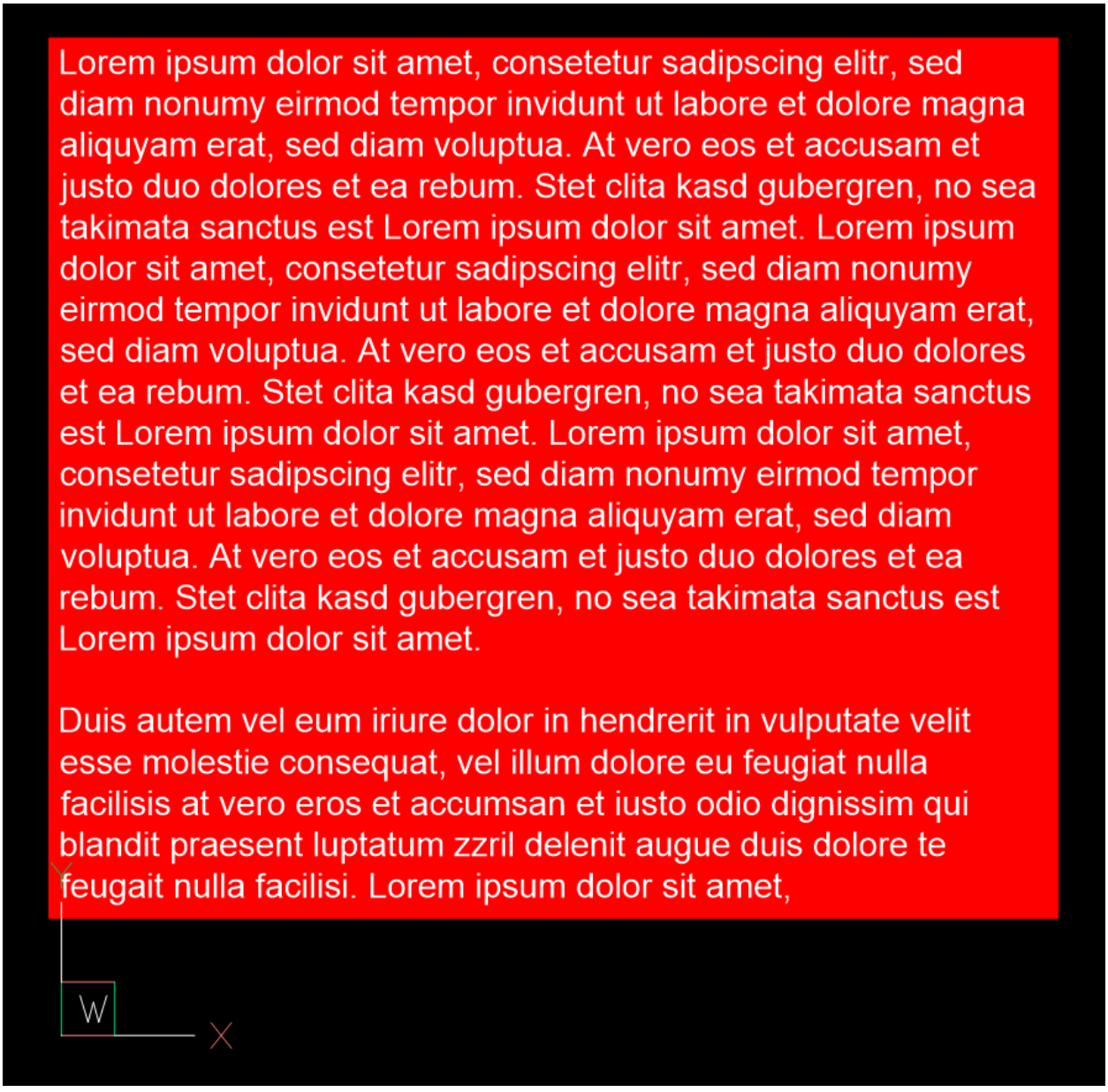
0	off
1	color defined by group code 63, 421 or 431
2	drawing window color
3	background (canvas) color
16	bit-flag text frame, see Open Design Alliance Specification 20.4.46

Group codes to define background fill attributes:

45	scaling factor for the border around the text, the value should be in the range of [1, 5], where 1 fits exact the MText entity
63	set the background color by *ACI*.
421	set the background color as *true-color* value.
431	set the background color by color name - no idea how this works
441	set the transparency of the background fill, not supported by AutoCAD or BricsCAD.

Group codes 45, 90 and 63 are required together if one of them is used. The group code 421 and 431 also requires the group code 63, even this value is ignored.

```
... <snip>
1 <str> eu feugiat nulla facilisis at vero eros et accumsan et iusto ...
73 <int> 1
44 <float> 1.0
90 <int> 1, b00000001   <<< use a color
63 <int> 1              <<< ACI color (red)
45 <float> 1.5          <<< bg scaling factor, relative to the char height
441 <int> 0             <<< ignored (optional)
... <snip>
```



The background scaling does not alter the `width`, `column_width` or `total_width` attributes. The background acquires additional space around the MTEXT entity.

Columns with background color:

Text Frame

The MTEXT entity can have a text frame only, without a background filling, group code 90 has value 16. In this case all other background related tags are removed (45, 63, 421, 431, 441) and the scaling factor is 1.5 by default.

XDATA for Text Frame

This XDATA exist only if the text frame flag in group code 90 is set and for DXF version < R2018!

```
...  <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_TEXT_BORDERS_BEGIN
1070 <int> 80       <<< group code for repeated flags
1070 <int> 16       <<< repeated group code 90?
1070 <int> 46       <<< group code for scaling factor, which is fixed?
1040 <float> 1.5    <<< scaling factor
1070 <int> 81       <<< group code for repeated flow direction?
1070 <int> 1        <<< flow direction?
1070 <int> 5        <<< group code for a handle, multiple entries possible
1005 <hex> #A8      <<< handle to the LWPOLYLINE text frame
1070 <int> 5        <<< group code for next handle
1005 <hex> #A9      <<< next handle
...
1000 <str> ACAD_MTEXT_TEXT_BORDERS_END
```

Extra LWPOLYLINE Entity as Text Frame

The newer versions of AutoCAD and BricsCAD get all the information they need from the MTEXT entity, but it seems that older versions could not handle the text frame property correct. Therefore AutoCAD and BricsCAD create a separated LWPOLYLINE entity for the text frame for DXF versions < R2018. The handle to this text frame entity is stored in the XDATA as group code 1005, see section above.

Because this LWPOLYLINE is not required *ezdxf* does **not** create such a text frame entity nor the associated XDATA and *ezdxf* also **removes** this data from loaded DXF files at the second loading stage.

Column Support

CAD applications build multiple columns by linking 2 or more MTEXT entities together. In this case each column is a self-sufficient entity in DXF version R13 until R2013. The additional columns specifications are stored in the XDATA if the MTEXT which represents the first column.

DXF R2018 changed the implementation into a single MTEXT entity which contains all the content text at once and stores the column specification in an embedded object.

Hint: The `width` attribute for the linked MTEXT entities could be wrong. Always use the `column_width` and the `total_width` attributes in column mode.

There are two column types, the **static** type has the same column height for all columns, the **dynamic** type can have the same (auto) height or an individual height for each column.

Common facts about columns for all column types:

- all columns have the same column width
- all columns have the same gutter width
- the top of the column are at the same height

Column Type

The column type defines how a CAD application should create the columns, this is not important for the file format, because the result of this calculation, the column count and the column height, is stored the DXF file.

Column Type in BricsCAD	Description
Static	All columns have the same height. The "auto height" flag is 0.
Dynamic (auto height)	Same as the static type, all columns have the same height. The "auto height" flag is 1. The difference to the static type is only important for interactive CAD applications.
Dynamic (manual height)	same as the dynamic (auto height) type, but each column can have an individual height.
No column	A regular MTEXT with "defined column height" attribute?

Column Type	Defined Height	Auto Height	Column Heights
Static	stored	False	not stored
Dynamic auto	stored	True	not stored
Dynamic manual	not stored	False	stored (last=0)

Column Count

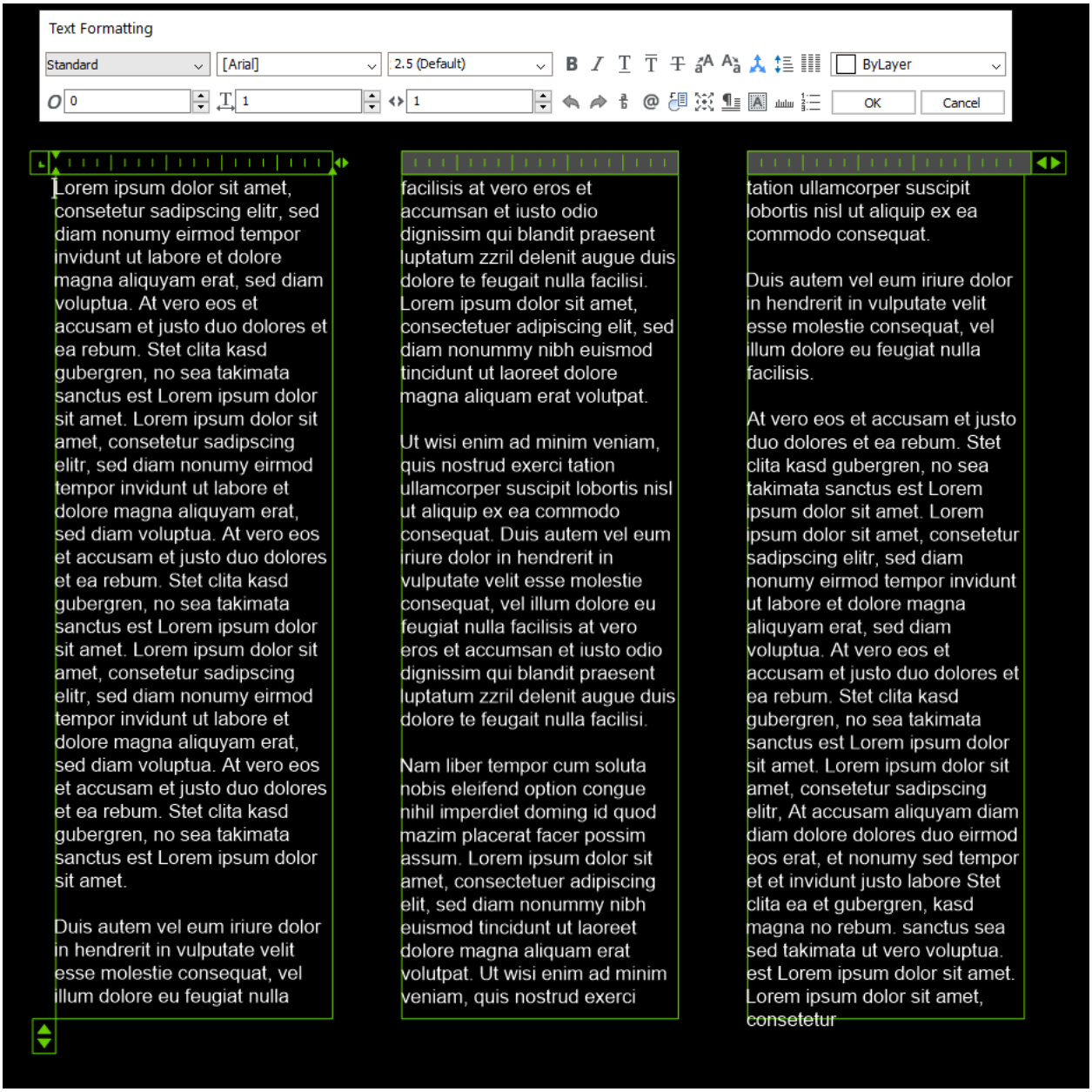
For DXF versions < R2018 the column count is always given by the count of linked MTEXT columns. Caution: the column count stored in the XDATA section by group code 76 may not match the count of linked MTEXT entities and AutoCAD is OK with that! In DXF R2018+ this property is not available, because there are no linked MTEXT entities anymore.

R2018+: For the column types "static" and "dynamic manual" the correct column count is stored as group code 72. For the column type "dynamic auto" the stored *column count is 0*. It is possible to calculate the column count from the total width and the column width if the total width is correct like in AutoCAD and BricsCAD.

Static Columns R2000

Example for a **static** column specification:

- Column Type: Static
- Number of Columns: 3
- Height: 150.0, manual entered value and all columns have the same height
- Width: 50.0
- Gutter Width: 12.5



The column height is stored as the “defined column height” in XDATA (46) or the embedded object (41).

DXF R2000 example with a static column specification stored in XDATA:

```
0
MTEXT
5           <<< entity handle
9D
102
{ACAD_XDICTIONARY
360
9F
102
}
330         <<< block record handle of owner layout
```

(continues on next page)

(continued from previous page)

```
1F
100
AcDbEntity
8           <<< layer
0
100         <<< begin of MTEXT specific data
AcDbMText
10          <<< (10, 20, 30) insert location in WCS
285.917876152751
20
276.101821192053
30
0.0
40          <<< character height in drawing units
2.5
41          <<< reference column width, if not in column mode
62.694...   <<< in column mode: the real column is defined in XDATA (48)
71          <<< attachment point
1
72          <<< text flow direction
1
3           <<< begin of text
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam ...
3
kimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit ...
3
ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ...
3
At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd ...
3
ore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio ...
1           <<< last text line and end of text
euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.
73          <<< line spacing style
1
44          <<< line spacing factor
1.0
1001
AcadAnnotative
1000
AnnotativeData
1002
{
1070
1
1070
0
1002
}
1001        <<< AppID "ACAD" contains the column specification
ACAD
1000
ACAD_MTEXT_COLUMN_INFO_BEGIN
1070
75          <<< group code column type
1070
1           <<< column type: 0=no column; 1=static columns; 2=dynamic columns
```

(continues on next page)

(continued from previous page)

```
1070
79          <<< group code column auto height
1070
0           <<< flag column auto height
1070
76          <<< group code column count
1070
3           <<< column count
1070
78          <<< group code column flow reversed
1070
0           <<< flag column flow reversed
1070
48          <<< group code column width
1040
50.0        <<< column width in column mode
1070
49          <<< group code column gutter
1040
12.5        <<< column gutter width
1000
ACAD_MTEXT_COLUMN_INFO_END
1000        <<< linked MTEXT entities specification
ACAD_MTEXT_COLUMNS_BEGIN
1070
47          <<< group code for column count, incl. the 1st column - this entity
1070
3           <<< column count
1005
1B4         <<< handle to 2nd column as MTEXT entity
1005
1B5         <<< handle to 3rd column as MTEXT entity
1000
ACAD_MTEXT_COLUMNS_END
1000
ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070
46          <<< group code for defined column height
1040
150.0       <<< defined column height
1000
ACAD_MTEXT_DEFINED_HEIGHT_END
```

The linked column MTEXT `#1B4` in a compressed representation:

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (348.417876152751, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0        <<< invalid reference column width
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46           <<< defined column height
1040 <float> 150.0
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

The linked MTEXT has no column specification except the "defined column height" in the XDATA. The reference column `width` is not the real value of 50.0, see XDATA group code 48 in the main MTEXT `#9D`, instead the total width of 175.0 is stored at group code 41. This is problem if a renderer try to render this MTEXT as a standalone entity. The renderer has to fit the content into the column width by itself and without the correct column width, this will produce an incorrect result.

There exist no back link to the main MTEXT `#9D`. The linked MTEXT entities appear after the main MTEXT in the layout space, but there can be other entities located between these linked MTEXT entities.

The linked column MTEXT `#1B5`:

```
0 <ctrl> MTEXT
5 <hex> #1B5
... <snip>
100 <ctrl> AcDbMText
10 <point> (410.917876152751, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0        <<< invalid reference column width
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46           <<< defined column height
1040 <float> 150.0
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

Static Columns R2018

The MTEXT entity in DXF R2018 contains all column information in a single entity. The text content of all three columns are stored in a continuous text string, the separation into columns has to be done by the renderer. The manual column break `\N` is **not** used to indicate automatic column breaks. The MTEXT renderer has to replicate the AutoCAD/BricsCAD rendering as exact as possible to achieve the same results, which is very hard without rendering guidelines or specifications.

The example from above in DXF R2018 with a static column specification stored in an embedded object:

```
0
MTEXT
5           <<< entity handle
9D
102
{ACAD_XDICTIONARY
360
9F
102
}
330         <<< block record handle of owner layout
1F
100
AcDbEntity
8           <<< layer
0
100
AcDbMText
10          <<< (10, 20, 30) insert location in WCS
285.917876152751
20
276.101821192053
30
```

(continues on next page)

(continued from previous page)

```
0.0
40          <<< character height in drawing units
2.5
41          <<< reference column width, if not in column mode
62.694536423841
46          <<< defined column height
150.0
71          <<< attachment point
1
72          <<< text flow direction
1
3           <<< text content of all three columns
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam n...
3
imata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit...
3
a rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lor...
3
vero eos et accusam et justo duo dolores et ea rebum. Stet clita ka...
3
eu feugiat nulla facilisis at vero eros et accumsan et iusto odio s...
3
od tincidunt ut laoreet dolore magna aliquam erat volutpat.   \P\PU...
3
e velit esse molestie consequat, vel illum dolore eu feugiat nulla ...
3
obis eleifend option congue nihil imperdiet doming id quod mazim pl...
3
m ad minim veniam, quis nostrud exerci tation ullamcorper suscipit ...
3
lisis.   \P\PAt vero eos et accusam et justo duo dolores et ea rebu...
3
t labore et dolore magna aliquyam erat, sed diam voluptua. At vero ...
3
litr, At accusam aliquyam diam diam dolore dolores duo eirmod eos e...
1
ipsum dolor sit amet, consetetur
73          <<< line spacing style
1
44          <<< line spacing factor
1.0
101         <<< column specification as embedded object
Embedded Object
70          <<< ???
1
10          <<< (10, 20, 30) text direction vector (local x-axis)
1.0
20
0.0
30
0.0
11          <<< (11, 21, 31) repeated insert location of AcDbMText
285.917876152751
21
276.101821192053
31
0.0
```

(continues on next page)

(continued from previous page)

```
40          <<< repeated reference column width
62.694536423841
41          <<< repeated defined column height
150.0
42          <<< extents (total) width
175.0
43          <<< extents (total) height, max. height if different column heights
150.0
71          <<< column type: 0=no column; 1=static columns; 2=dynamic columns
1
72          <<< column height count
3
44          <<< column width
50.0
45          <<< column gutter width
12.5
73          <<< flag column auto height
0
74          <<< flag reversed column flow
0
1001
AcadAnnotative
1000
AnnotativeData
1002
{
1070
1
1070
0
1002
}
```

Dynamic (auto height) Columns R2000

Example for a **dynamic** column specification:

- Column Type: Dynamic
- Number of Columns: 3
- Height: 158.189… adjusted by widget and all columns have the same height
- Width: 50.0
- Gutter Width: 12.5

```
0 <ctrl> MTEXT
5 <hex> #A2                 <<< entity handle
... <snip>
330 <hex> #1F               <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0                   <<< layer
100 <ctrl> AcDbMText
10 <point> (-133.714579865783, 276.101821192053, 0.0)   <<< insert location in WCS
40 <float> 2.5              <<< character height in drawing units
41 <float> 62.694536423841  <<< reference column width, if not in column mode
```

(continues on next page)

(continued from previous page)

```
71 <int> 1                  <<< attachment point
72 <int> 1                  <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed dia...
... <snip>
73 <int> 1                  <<< line spacing style
44 <float> 1.0              <<< line spacing factor
1001 <ctrl> AcadAnnotative
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_COLUMN_INFO_BEGIN
1070 <int> 75               <<< column type: 2=dynamic columns
1070 <int> 2
1070 <int> 79               <<< flag column auto height
1070 <int> 1
1070 <int> 76               <<< column count
1070 <int> 3
1070 <int> 78               <<< flag column flow reversed
1070 <int> 0
1070 <int> 48               <<< column width in column mode
1040 <float> 50.0
1070 <int> 49               <<< column gutter width
1040 <float> 12.5
1000 <str> ACAD_MTEXT_COLUMN_INFO_END
1000 <str> ACAD_MTEXT_COLUMNS_BEGIN
1070 <int> 47               <<< column count
1070 <int> 3
1005 <hex> #1B6             <<< handle to 2. column as MTEXT entity
1005 <hex> #1B7             <<< handle to 3. column as MTEXT entity
1000 <str> ACAD_MTEXT_COLUMNS_END
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46               <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

The linked column MTEXT `#1B6`:

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-71.214579865783, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0    <<< invalid column width
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46       <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

The linked column MTEXT `#1B7`:

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-8.714579865783, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0    <<< invalid column width
```

(continues on next page)

(continued from previous page)

```
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46       <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

Dynamic (auto height) Columns R2018

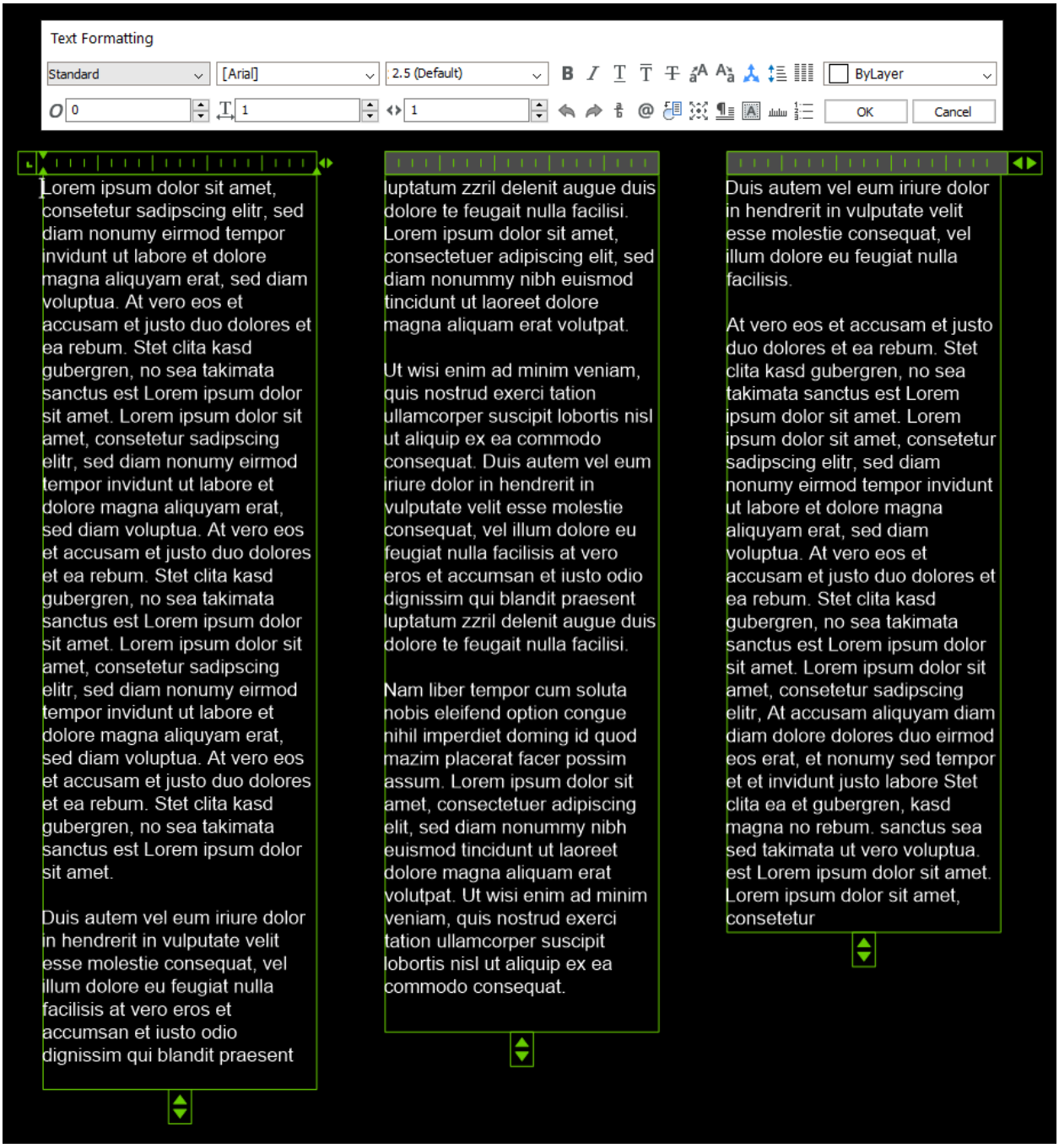
```
0 <ctrl> MTEXT
5 <hex> #A2                     <<< entity handle
102 <ctrl> {ACAD_XDICTIONARY
360 <hex> #A3
102 <ctrl> }
330 <hex> #1F                   <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0                       <<< layer
100 <ctrl> AcDbMText
10 <point> (-133.714579865783, 276.101821192053, 0.0)   <<< insert location in WCS
40 <float> 2.5                  <<< character height in drawing units
41 <float> 62.694536423841      <<< reference column width, if not in column mode
46 <float> 158.189308131867     <<< defined column height
71 <int> 1                      <<< attachment point
72 <int> 1                      <<< text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1                      <<< line spacing style
44 <float> 1.0                  <<< line spacing factor
101 <ctrl> Embedded Object
70 <int> 1, b00000001           <<< ???
10 <point> (1.0, 0.0, 0.0)      <<< text direction vector (local x-axis)
11 <point> (-133.714579865783, 276.101821192053, 0.0)   <<< repeated insert location
40 <float> 62.694536423841      <<< repeated reference column width
41 <float> 158.189308131867     <<< repeated defined column height
42 <float> 175.0                <<< extents (total) width
43 <float> 158.189308131867     <<< extents (total) height, max. height if different␣
↪column heights
71 <int> 2                      <<< column type: 2=dynamic columns
72 <int> 0                      <<< column height count
44 <float> 50.0                 <<< column width
45 <float> 12.5                 <<< column gutter width
73 <int> 1                      <<< flag column auto height
74 <int> 0                      <<< flag reversed column flow
1001 <ctrl> AcadAnnotative
1000 <str> AnnotativeData
1002 <str> {
1070 <int> 1
1070 <int> 0
1002 <str> }
```

Dynamic (manual height) Columns R2000

Example for a **dynamic** column specification with manual height definition for three columns with different column heights. None of the (linked) MTEXT entities does contain XDATA for the defined column height.

Hint: If "content type" is 2 and flag "column auto height" is 0, no defined height in XDATA.

- Column Type: Dynamic
- Number of Columns: 3
- Height: 164.802450331126, max. column height
- Width: 50.0
- Gutter Width: 12.5




```
0 <ctrl> MTEXT
5 <hex> #9C                     <<< entity handle
330 <hex> #1F                   <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0                       <<< layer
100 <ctrl> AcDbMText
10 <point> (69.806121185863, 276.101821192053, 0.0)     <<< insert location in WCS
40 <float> 2.5                  <<< character height in drawing units
41 <float> 62.694536423841      <<< reference column width, if not in column mode
71 <int> 1                      <<< attachment point
72 <int> 1                      <<< flag text flow direction
```

(continues on next page)

(continued from previous page)

```
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, ...
... <snip>
73 <int> 1                      <<< line spacing style
44 <float> 1.0                  <<< line spacing factor
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_COLUMN_INFO_BEGIN
1070 <int> 75                   <<< column type: 2=dynamic columns
1070 <int> 2
1070 <int> 79                   <<< flag column auto height
1070 <int> 0
1070 <int> 76                   <<< column count
1070 <int> 3
1070 <int> 78                   <<< flag column flow reversed
1070 <int> 0
1070 <int> 48                   <<< column width in column mode
1040 <float> 50.0
1070 <int> 49                   <<< column gutter width
1040 <float> 12.5
1070 <int> 50                   <<< column height count
1070 <int> 3
1040 <float> 164.802450331126   <<< column height 1. column
1040 <float> 154.311699779249   <<< column height 2. column
1040 <float> 0.0                <<< column height 3. column, takes the rest?
1000 <str> ACAD_MTEXT_COLUMN_INFO_END
1000 <str> ACAD_MTEXT_COLUMNS_BEGIN
1070 <int> 47                   <<< column count
1070 <int> 3
1005 <hex> #1B2                 <<< handle to 2. column as MTEXT entity
1005 <hex> #1B3                 <<< handle to 3. column as MTEXT entity
1000 <str> ACAD_MTEXT_COLUMNS_END
```

The linked column MTEXT `#1B2`:

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (132.306121185863, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0            <<< invalid reference column width
... <snip>
73 <int> 1
44 <float> 1.0
```

The linked column MTEXT `#1B3`:

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (194.806121185863, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0            <<< invalid reference column width
... <snip>
73 <int> 1
44 <float> 1.0
```

Dynamic (manual height) Columns R2018

Hint: If "content type" is 2 and flag "column auto height" is 0, the "defined column height" is 0.0.

```
0 <ctrl> MTEXT
5 <hex> #9C                     <<< entity handle
330 <hex> #1F
100 <ctrl> AcDbEntity
8 <str> 0                       <<< block record handle of owner layout
100 <ctrl> AcDbMText
10 <point> (69.806121185863, 276.101821192053, 0.0)     <<< insert location in WCS
40 <float> 2.5                  <<< character height in drawing units
41 <float> 62.694536423841      <<< reference column width, if not in column mode
46 <float> 0.0                  <<< defined column height
71 <int> 1                      <<< attachment point
72 <int> 1                      <<< text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1                      <<< line spacing style
44 <float> 1.0                  <<< line spacing factor
101 <ctrl> Embedded Object
70 <int> 1, b00000001           <<< ???
10 <point> (1.0, 0.0, 0.0)      <<< text direction vector (local x-axis)
11 <point> (69.806121185863, 276.101821192053, 0.0)     <<< repeated insert location
40 <float> 62.694536423841      <<< repeated reference column width
41 <float> 0.0                  <<< repeated defined column height
42 <float> 175.0                <<< extents (total) width
43 <float> 164.802450331126     <<< extents (total) height, max. height if different␣
↪column heights
71 <int> 2                      <<< column type: 2=dynamic columns
72 <int> 3                      <<< column height count
44 <float> 50.0                 <<< column width
45 <float> 12.5                 <<< column gutter width
73 <int> 0                      <<< flag column auto height
74 <int> 0                      <<< flag reversed column flow
46 <float> 164.802450331126     <<< column height 1. column
46 <float> 154.311699779249     <<< column height 2. column
46 <float> 0.0                  <<< column height 3. column, takes the rest?
```

No Columns R2000

I have no idea why this column type exist, but at least provides a reliable value for the MTEXT height by the "defined column height" attribute. The column type is not stored in the MTEXT entity and is therefore not detectable!

- Column Type: No columns
- Number of Columns: 1
- Height: 158.189308131867, defined column height
- Width: 175.0, reference column width

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
```

(continues on next page)

(continued from previous page)

```
10 <point> (-344.497343455795, 276.101821192053, 0.0)   <<< insert location in WCS
40 <float> 2.5          <<< character height in drawing units
41 <float> 175.0        <<< reference column width
71 <int> 1              <<< attachment point
72 <int> 1              <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1              <<< line spacing style
44 <float> 1.0          <<< line spacing factor
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46           <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

No Columns R2018

Does not contain an embedded object.

```
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-334.691900433414, 276.101821192053, 0.0)   <<< insert location in WCS
40 <float> 2.5                  <<< character height in drawing units
41 <float> 175.0                <<< reference column width
46 <float> 158.189308131867     <<< defined column height
71 <int> 1                      <<< attachment point
72 <int> 1                      <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, ...
... <snip>
73 <int> 1                      <<< line spacing style
44 <float> 1.0                  <<< line spacing factor
1001 <ctrl> AcadAnnotative
... <snip>
```

DXF Objects

TODO

6.13.3 Management Structures

Block Management Structures

A BLOCK is a layout like the modelspace or a paperspace layout, with the similarity that all these layouts are containers for graphical DXF entities. This block definition can be referenced in other layouts by the INSERT entity. By using block references, the same set of graphical entities can be located multiple times at different layouts, this block references can be stretched and rotated without modifying the original entities. A block is referenced only by its name defined by the DXF tag `(2, name)`, there is a second DXF tag `(3, name2)` for the block name, which is not further documented by Autodesk, just ignore it.

The `(10, base_point)` tag (in BLOCK defines a insertion point of the block, by 'inserting' a block by the INSERT entity, this point of the block is placed at the location defined by the `(10, insert)` tag in the INSERT entity, and it is also the base point for stretching and rotation.

A block definition can contain INSERT entities, and it is possible to create cyclic block definitions (a BLOCK contains a INSERT of itself), but this should be avoided, CAD applications will not load the DXF file at all or maybe just crash. This is also the case for all other kinds of cyclic definitions like: BLOCK "A" -> INSERT BLOCK "B" and BLOCK "B" -> INSERT BLOCK "A".

See also:

- ezdxf DXF Internals: *BLOCKS Section*
- DXF Reference: BLOCKS Section
- DXF Reference: BLOCK Entity
- DXF Reference: ENDBLK Entity
- DXF Reference: INSERT Entity

Block Names

Block names has to be unique and they are case insensitive ("Test" == "TEST"). If there are two or more block definitions with the same name, AutoCAD merges these blocks into a single block with unpredictable properties of all these blocks. In my test with two blocks, the final block has the name of the first block and the base-point of the second block, and contains all entities of both blocks.

Block Definitions in DXF R12

In DXF R12 the definition of a block is located in the BLOCKS section, no additional structures are needed. The definition starts with a BLOCK entity and ends with a ENDBLK entity. All entities between this two entities are the content of the block, the block is the owner of this entities like any layout.

As shown in the DXF file below (created by AutoCAD LT 2018), the BLOCK entity has no handle, but ezdxf writes also handles for the BLOCK entity and AutoCAD doesn't complain.

DXF R12 BLOCKS structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of BLOCKS section
BLOCKS
...         <<< modelspace and paperspace block definitions not shown,
...         <<< see layout management
...
0           <<< start of a BLOCK definition
BLOCK
8           <<< layer
0
2           <<< block name
ArchTick
70          <<< flags
1
10          <<< base point, x
0.0
20          <<< base point, y
```

(continues on next page)

(continued from previous page)

```
0.0
30          <<< base point, z
0.0
3           <<< second BLOCK name, same as (2, name)
ArchTick
1           <<< xref name, if block is an external reference
            <<< empty string!
0           <<< start of the first entity of the BLOCK
LINE
5
28E
8
0
62
0
10
500.0
20
500.0
30
0.0
11
500.0
21
511.0
31
0.0
0           <<< start of the second entity of the BLOCK
LINE
...
0.0
0           <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5           <<< ENDBLK gets a handle by AutoCAD, but BLOCK didn't
2F2
8           <<< as every entity, also ENDBLK requires a layer (same as BLOCK entity!)
0
0           <<< start of next BLOCK entity
BLOCK
...
0           <<< end BLOCK entity
ENDBLK
0           <<< end of BLOCKS section
ENDSEC
```

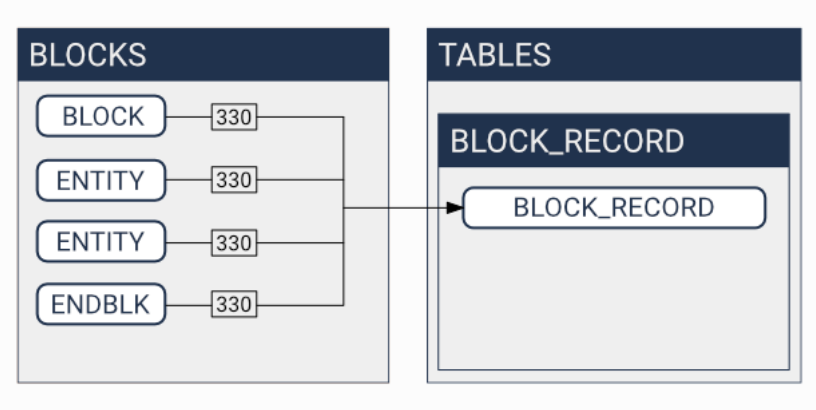
Block Definitions in DXF R2000+

The overall organization in the BLOCKS sections remains the same, but additional tags in the BLOCK entity, have to be maintained.

Especially the concept of ownership is important. Since DXF R13 every graphic entity is associated to a specific layout and a BLOCK definition is also a layout. So all entities in the BLOCK definition, including the BLOCK and the ENDBLK entities, have an owner tag `(330, ...)`, which points to a BLOCK_RECORD entry in the BLOCK_RECORD table. This BLOCK_RECORD is the main management structure for all layouts and is the real owner of the layout entities.

As you can see in the chapter about *Layout Management Structures*, this concept is also valid for modelspace and pa-

perspace layouts, because these layouts are also BLOCKS, with the special difference, that the entities of the modelspace and the *active* paperspace layout are stored in the ENTITIES section.



See also:

- *DXF R13 and later tag structure*
- ezdxf DXF Internals: *TABLES Section*
- DXF Reference: TABLES Section
- DXF Reference: BLOCK_RECORD Entity

DXF R13 BLOCKS structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of BLOCKS section
BLOCKS
...         <<< modelspace and paperspace block definitions not shown,
...         <<< see layout management
0           <<< start of BLOCK definition
BLOCK
5           <<< even BLOCK gets a handle now ;)
23A
330         <<< owner tag, the owner of a BLOCK is a BLOCK_RECORD in the
...         BLOCK_RECORD table
238
100         <<< subclass marker
AcDbEntity
8           <<< layer of the BLOCK definition
0
100         <<< subclass marker
AcDbBlockBegin
2           <<< BLOCK name
ArchTick
70          <<< flags
0
10          <<< base point, x
0.0
20          <<< base point, y
0.0
30          <<< base point, z
0.0
3           <<< second BLOCK name, same as (2, name)
```

(continues on next page)

(continued from previous page)

```
ArchTick
1           <<< xref name, if block is an external reference
            <<< empty string!
0           <<< start of the first entity of the BLOCK
LWPOLYLINE
5
239
330         <<< owner tag of LWPOLYLINE
238         <<< handle of the BLOCK_RECORD!
100
AcDbEntity
8
0
6
ByBlock
62
0
100
AcDbPolyline
90
2
70
0
43
0.15
10
-0.5
20
-0.5
10
0.5
20
0.5
0           <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5           <<< handle
23B
330         <<< owner tag, same BLOCK_RECORD as for the BLOCK entity
238
100         <<< subclass marker
AcDbEntity
8           <<< ENDBLK requires the same layer as the BLOCK entity!
0
100         <<< subclass marker
AcDbBlockEnd
0           <<< start of the next BLOCK
BLOCK
...
0
ENDBLK
...
0           <<< end of the BLOCKS section
ENDSEC
```

DXF R13 BLOCK_RECORD structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of TABLES section
TABLES
0           <<< start of a TABLE
TABLE
2           <<< start of the BLOCK_RECORD table
BLOCK_RECORD
5           <<< handle of the table
1
330         <<< owner tag of the table
0           <<< is always #0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of table entries, not reliable
4
0           <<< start of first BLOCK_RECORD entry
BLOCK_RECORD
5           <<< handle of BLOCK_RECORD, in ezdxf often referred to as "layout key"
1F
330         <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbBlockTableRecord
2           <<< name of the BLOCK or LAYOUT
*Model_Space
340         <<< pointer to the associated LAYOUT object
4AF
70          <<< AC1021 (R2007) block insertion units
0
280         <<< AC1021 (R2007) block explodability
1
281         <<< AC1021 (R2007) block scalability
0

...         <<< paperspace not shown
...
0           <<< next BLOCK_RECORD
BLOCK_RECORD
5           <<< handle of BLOCK_RECORD, in ezdxf often referred to as "layout key"
238
330         <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbBlockTableRecord
2           <<< name of the BLOCK
ArchTick
340         <<< pointer to the associated LAYOUT object
0           <<< #0, because BLOCK doesn't have an associated LAYOUT object
70          <<< AC1021 (R2007) block insertion units
0
280         <<< AC1021 (R2007) block explodability
1
```

(continues on next page)

(continued from previous page)

```
281          <<< AC1021 (R2007) block scalability
0
0            <<< end of BLOCK_RECORD table
ENDTAB
0            <<< next TABLE
TABLE
...
0
ENDTAB
0            <<< end of TABLES section
ENDESC
```

Layout Management Structures

Layouts are separated entity spaces, there are three different Layout types:

1. modelspace contains the 'real' world representation of the drawing subjects in real world units.
2. paperspace layouts are used to create different drawing sheets of the modelspace subjects for printing or PDF export
3. Blocks are reusable sets of graphical entities, inserted/referenced by the INSERT entity.

All layouts have at least a BLOCK definition in the BLOCKS section and since DXF R13 exist the BLOCK_RECORD table with an entry for every BLOCK in the BLOCKS section.

See also:

Information about *Block Management Structures*

The name of the modelspace BLOCK is "*Model_Space" (DXF R12: "$MODEL_SPACE") and the name of the *active* paperspace BLOCK is "*Paper_Space" (DXF R12: "$PAPER_SPACE"), the entities of these two layouts are stored in the ENTITIES section, DXF R12 supports just one paperspace layout.

DXF R13+ supports multiple paperspace layouts, the *active* layout is still called "*Paper_Space", the additional *inactive* paperspace layouts are named by the scheme "*Paper_Spacennnn", where the first inactive paper space is called "*Paper_Space0", the second "*Paper_Space1" and so on. A none consecutive numbering is tolerated by AutoCAD. The content of the inactive paperspace layouts are stored as BLOCK content in the BLOCKS section. These names are just the DXF internal layout names, each layout has an additional layout name which is displayed to the user by the CAD application.

A BLOCK definition and a BLOCK_RECORD is not enough for a proper layout setup, an LAYOUT entity in the OBJECTS section is also required. All LAYOUT entities are managed by a DICTIONARY entity, which is referenced as "ACAD_LAYOUT" entity in the root DICTIONARY of the DXF file.

Note: All floating point values are rounded to 2 decimal places for better readability.

LAYOUT Entity

Since DXF R2000 modelspace and paperspace layouts require the DXF LAYOUT entity.

```
0
LAYOUT
5       <<< handle
59
102     <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1C3
102
}
102     <<< reactor (required?)
{ACAD_REACTORS
330
1A      <<< pointer to "ACAD_LAYOUT" DICTIONARY (layout management table)
102
}
330     <<< owner handle
1A      <<< pointer to "ACAD_LAYOUT" DICTIONARY (same as reactor pointer)
100     <<< PLOTSETTINGS
AcDbPlotSettings
1       <<< page setup name

2       <<< name of system printer or plot configuration file
none_device
4       <<< paper size, part in braces should follow the schema
...     (width_x_height_unit) unit is 'Inches' or 'MM'
...     Letter\_(8.50_x_11.00_Inches)  the part in front of the braces is
...     ignored by AutoCAD
6       <<< plot view name

40      <<< size of unprintable margin on left side of paper in millimeters,
...     defines also the plot origin-x
6.35
41      <<< size of unprintable margin on bottom of paper in millimeters,
...     defines also the plot origin-y
6.35
42      <<< size of unprintable margin on right side of paper in millimeters
6.35
43      <<< size of unprintable margin on top of paper in millimeters
6.35
44      <<< plot paper size: physical paper width in millimeters
215.90
45      <<< plot paper size: physical paper height in millimeters
279.40
46      <<< X value of plot origin offset in millimeters, moves the plot origin-x
0.0
47      <<< Y value of plot origin offset in millimeters, moves the plot origin-y
0.0
48      <<< plot window area: X value of lower-left window corner
0.0
49      <<< plot window area: Y value of lower-left window corner
0.0
140     <<< plot window area: X value of upper-right window corner
```

(continues on next page)

(continued from previous page)

```
0.0
141     <<< plot window area: Y value of upper-right window corner
0.0
142     <<< numerator of custom print scale: real world (paper) units, 1.0
...     for scale 1:50
1.0
143     <<< denominator of custom print scale: drawing units, 50.0
...     for scale 1:50
1.0
70      <<< plot layout flags, bit-coded (... too many options)
688     <<< b1010110000 = UseStandardScale(16)/PlotPlotStyle(32)
...     PrintLineweights(128)/DrawViewportsFirst(512)
72      <<< plot paper units (0/1/2 for inches/millimeters/pixels), are
...     pixels really supported?
0
73      <<< plot rotation (0/1/2/3 for 0deg/90deg counter-cw/upside-down/90deg cw)
1       <<< 90deg clockwise
74      <<< plot type 0-5 (... too many options)
5       <<< 5 = layout information
7       <<< current plot style name, e.g. 'acad.ctb' or 'acadlt.ctb'

75      <<< standard scale type 0-31 (... too many options)
16      <<< 16 = 1:1, also 16 if user scale type is used
147     <<< unit conversion factor
1.0     <<< for plot paper units in mm, else  0.03937... (1/25.4) for inches
...     as plot paper units
76      <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0       <<< as displayed
77      <<< shade plot resolution level 1-5 (... too many options)
2       <<< normal
78      <<< shade plot custom DPI: 100-32767, Only applied when shade plot
...     resolution level is set to 5 (Custom)
300
148     <<< paper image origin: X value
0.0
149     <<< paper image origin: Y value
0.0
100     <<< LAYOUT settings
AcDbLayout
1       <<< layout name
Layout1
70      <<< flags bit-coded
1       <<< 1 = Indicates the PSLTSCALE value for this layout when this
...     layout is current
71      <<< Tab order ("Model" tab always appears as the first tab
...     regardless of its tab order)
1
10      <<< minimum limits for this layout (defined by LIMMIN while this
...     layout is current)
-0.25   <<<     x value, distance of the left paper margin from the plot
...     origin-x, in plot paper units and by scale (e.g. x50 for 1:50)
20      <<<     group code for y value
-0.25   <<<     y value, distance of the bottom paper margin from the plot
...     origin-y, in plot paper units and by scale (e.g. x50 for 1:50)
11      <<< maximum limits for this layout (defined by LIMMAX while this
...     layout is current)
10.75   <<<     x value, distance of the right paper margin from the plot
```

(continues on next page)

(continued from previous page)

```
...     origin-x, in plot paper units and by scale (e.g. x50 for 1:50)
21      <<<     group code for y value
8.25    <<<     y value, distance of the top paper margin from the plot
...     origin-y, in plot paper units and by scale (e.g. x50 for 1:50)
12      <<< insertion base point for this layout (defined by INSBASE while
...     this layout is current)
0.0     <<<     x value
22      <<<     group code for y value
0.0     <<<     y value
32      <<<     group code for z value
0.0     <<<     z value
14      <<< minimum extents for this layout (defined by EXTMIN while this
...     layout is current), AutoCAD default is (1e20, 1e20, 1e20)
1.05    <<<     x value
24      <<<     group code for y value
0.80    <<<     y value
34      <<<     group code for z value
0.0     <<<     z value
15      <<< maximum extents for this layout (defined by EXTMAX while this
...     layout is current), AutoCAD default is (-1e20, -1e20, -1e20)
9.45    <<<     x value
25      <<<     group code for y value
7.20    <<<     y value
35      <<<     group code for z value
0.0     <<<     z value
146     <<< elevation ???
0.0
13      <<< UCS origin (3D Point)
0.0     <<<     x value
23      <<<     group code for y value
0.0     <<<     y value
33      <<<     group code for z value
0.0     <<<     z value
16      <<< UCS X-axis (3D vector)
1.0     <<<     x value
26      <<<     group code for y value
0.0     <<<     y value
36      <<<     group code for z value
0.0     <<<     z value
17      <<< UCS Y-axis (3D vector)
0.0     <<<     x value
27      <<<     group code for y value
1.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
76      <<< orthographic type of UCS 0-6 (... too many options)
0       <<< 0 = UCS is not orthographic ???
330     <<< ID/handle of required block table record
58
331     <<< ID/handle to the viewport that was last active in this layout
...     when the layout was current
1B9
1001    <<< extended data (ignore)
...
```

And as it seems this is also not enough for a well defined LAYOUT, at least a “main” VIEWPORT entity with ID=1 is required for paperspace layouts, located in the entity space of the layout.

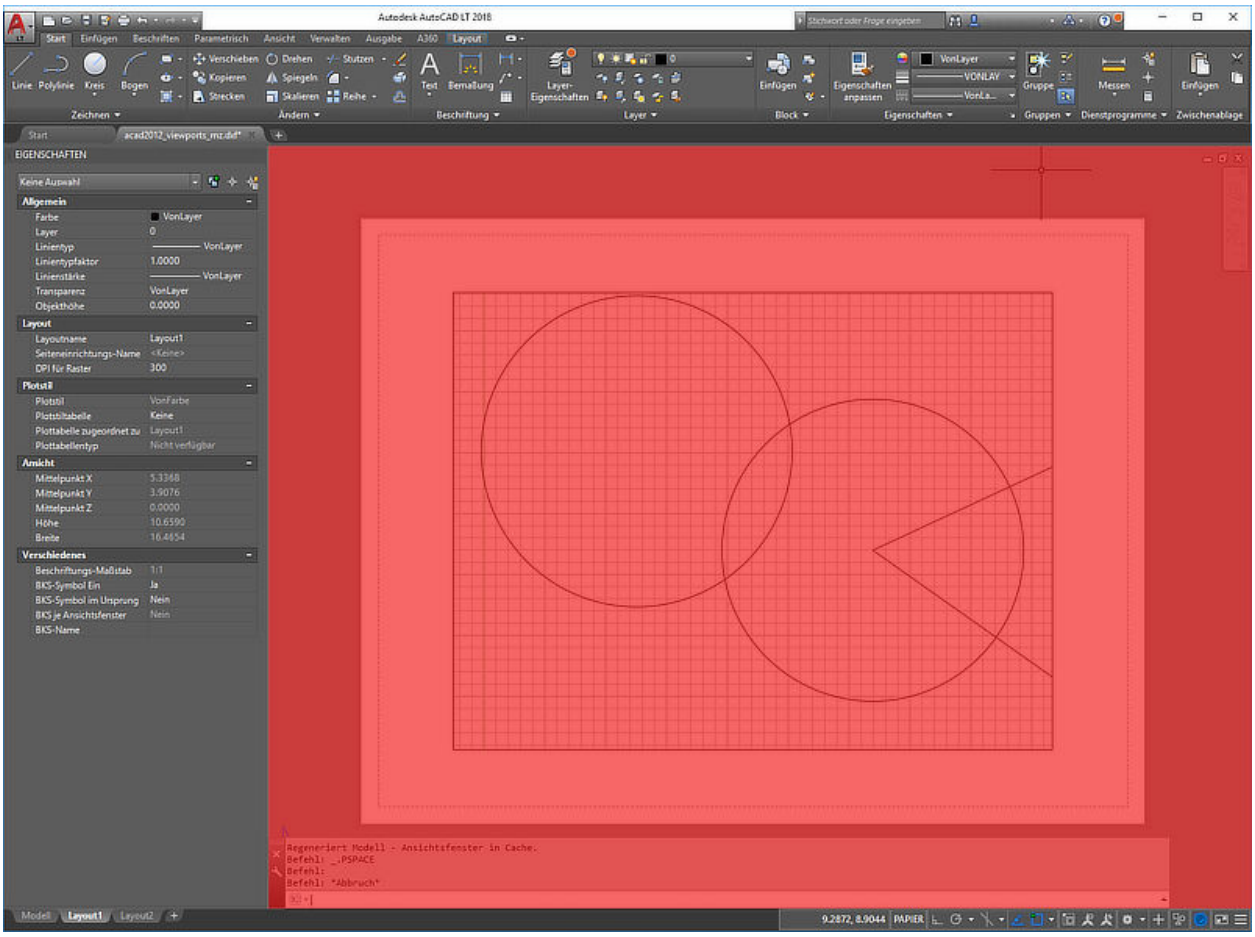
The modelspace layout requires (?) a VPORT entity in the VPORT table (group code 331 in the AcDbLayout subclass).

Main VIEWPORT Entity for LAYOUT

The "main" viewport for layout "Layout1" shown above. This viewport is located in the associated BLOCK definition called "*Paper_Space0". Group code 330 in subclass AcDbLayout points to the BLOCK_RECORD of "*Paper_Space0".

Remember: the entities of the *active* paperspace layout are located in the ENTITIES section, therefore "Layout1" is not the active paperspace layout.

The "main" VIEWPORT describes, how the application shows the paperspace layout on the screen, and I guess only AutoCAD needs this values.




```
0
VIEWPORT
5       <<< handle
1B4
102     <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1B5
102
}
330     <<< owner handle
58      <<< points to BLOCK_RECORD (same as group code 330 in AcDbLayout of
```

(continues on next page)

(continued from previous page)

```
...     "Layout1")
100
AcDbEntity
67      <<< paperspace flag
1       <<< 0 = modelspace; 1 = paperspace
8       <<< layer,
0
100
AcDbViewport
10      <<< Center point (in WCS)
5.25    <<<     x value
20      <<<     group code for y value
4.00    <<<     y value
30      <<<     group code for z value
0.0     <<<     z value
40      <<< width in paperspace units
23.55   <<< VIEW size in AutoCAD, depends on the workstation configuration
41      <<< height in paperspace units
9.00    <<< VIEW size in AutoCAD, depends on the workstation configuration
68      <<< viewport status field -1/0/n
2       <<< >0  On and active. The value indicates the order of stacking for
...     the viewports, where 1 is the active viewport, 2 is the next, and so forth
69      <<< viewport ID
1       <<< "main" viewport has always ID=1
12      <<< view center point in Drawing Coordinate System (DCS), defines
...     the center point of the VIEW in relation to the LAYOUT origin
5.25    <<<     x value
22      <<<     group code for y value
4.00    <<<     y value
13      <<< snap base point in modelspace
0.0     <<<     x value
23      <<<     group code for y value
0.0     <<<     y value
14      <<< snap spacing in modelspace units
0.5     <<<     x value
24      <<<     group code for y value
0.5     <<<     y value
15      <<< grid spacing in modelspace units
0.5     <<<     x value
25      <<<     group code for y value
0.5     <<<     y value
16      <<< view direction vector from target (in WCS)
0.0     <<<     x value
26      <<<     group code for y value
0.0     <<<     y value
36      <<<     group code for z value
1.0     <<<     z value
17      <<< view target point
0.0     <<<     x value
27      <<<     group code for y value
0.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
42      <<<     perspective lens length, focal length?
50.0    <<<     50mm
43      <<<     front clip plane z value
0.0     <<<     z value
```

(continues on next page)

(continued from previous page)

```
44      <<<     back clip plane z value
0.0     <<<     z value
45      <<<     view height (in modelspace units)
9.00
50      <<< snap angle
0.0
51      <<< view twist angle
0.0
72      <<< circle zoom percent
1000
90      <<< Viewport status bit-coded flags (... too many options)
819232  <<< b11001000000000100000
1       <<< plot style sheet name assigned to this viewport

281     <<< render mode (... too many options)
0       <<< 0 = 2D optimized (classic 2D)
71      <<< UCS per viewport flag
1       <<< 1 = This viewport stores its own UCS which will become the
...     current UCS whenever the viewport is activated
74      <<< Display UCS icon at UCS origin flag
0       <<< this field is currently being ignored and the icon always
...     represents the viewport UCS
110     <<< UCS origin (3D point)
0.0     <<<     x value
120     <<<     group code for y value
0.0     <<<     y value
130     <<<     group code for z value
0.0     <<<     z value
111     <<< UCS X-axis (3D vector)
1.0     <<<     x value
121     <<<     group code for y value
0.0     <<<     y value
131     <<<     group code for z value
0.0     <<<     z value
112     <<< UCS Y-axis (3D vector)
0.0     <<<     x value
122     <<<     group code for y value
1.0     <<<     y value
132     <<<     group code for z value
0.0     <<<     z value
79      <<< Orthographic type of UCS (... too many options)
0       <<< 0 = UCS is not orthographic
146     <<< elevation
0.0
170     <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0       <<< as displayed
61      <<< frequency of major grid lines compared to minor grid lines
5       <<< major grid subdivided by 5
348     <<< visual style ID/handle (optional)
9F
292     <<< default lighting flag, on when no user lights are specified.
1
282     <<< Default lighting type (0/1 = one distant light/two distant lights)
1       <<< one distant light
141     <<< view brightness
0.0
142     <<< view contrast
```

(continues on next page)

(continued from previous page)

```
0.0
63      <<< ambient light color (ACI), write only if not black color
250
421     <<< ambient light color (RGB), write only if not black color
3355443
```

6.13.4 Miscellaneous

Notes on Rendering DXF Content

A collection of AutoCAD behaviors determined experimentally. There may be mistakes and misunderstandings of the inner workings of the algorithms. Not all edge cases may have been considered.

Colors

- Most entities are colored contextually, based on the layer or block that they reside in.
- Usually entity colors are stored as AutoCAD Color Indices (ACI) as an index into a lookup table. Different CAD applications may use different color palettes making consistent coloring difficult.
- If a block insert is placed on layer 'A', and the block contains an entity on layer 'B' with BYLAYER color: the entity will be drawn with the color of layer 'B'.
- If a block insert is placed on layer 'A', and the block contains an entity on layer '0' with BYLAYER color: the entity will be drawn with the color of layer 'A', it seems that layer '0' is the only special case for this.
- Ff an entity has BYBLOCK color set, and it exists outside a block: it will take on the layout default color which is white in the modelspace and black in the paperspace.

Layers and Draw Order

- Layer names are case-insensitive, the document layer table keys are stored in lowercase, and in original style in all other use cases (e.g. *entity.dxf.layer*).
- Layers do not play a role in entity draw order, only whether they appear at all based on the visibility of the layer.
- It appears that Insert entities have a single element in terms of draw order
  - Entities inside a block can overlap each other and so have a draw order inside the block, but two Insert entities cannot interleave the contents of their blocks. One is completely drawn on top of the other.
- For entities inside a block, the visibility of the layer that the block is inserted does not affect the visibility of the entity *unless* the entity was created on layer 0 in which case the reverse is true:
  - scenario: block created containing entity A (layer 0) and entity B (layer 1). The block is inserted into layer 2
  - entity B visible if and only if layer 1 is visible
  - entity A visible if and only if layer 2 is visible

TEXT

- The anchor of single line TEXT entities (and ATTRIB entities) is *always* the left-baseline regardless of what alignment parameters are stored in the DXF. Those are for re-adjusting the anchor when the text is edited.
- Attrib entities can have formatting commands in them

MTEXT

- The *char_height* in DXF corresponds to the cap-height of the font.
- The default line spacing is 5/3 * cap-height between the previous baseline and the next baseline. The *line_space_factor* is a factor applied directly to this value, so a factor of 3/5 results in 0 space between lines, because each baseline is 1 * cap-height apart.
- The middle (vertical) justification of MTEXT entities seems to be midpoint between the x-height of the first line to the baseline of the last line.
- MTEXT word wrapping seems to only break on spaces, not underscores or dashes.
- MTEXT word wrapping seems to treat multiple spaces between lines as if they were a single space.
- Alignment seems to ignore extra spaces at the start or end of lines except for the first line, where spaces at the beginning of the string have an effect. Whitespace at the beginning of the text can trigger word wrapping, which creates a single blank line at the start
- If a line ends with an explicit newline character and is shorter than the column width, only one newline is inserted.
- If a line is a single word wider than the column width, it will not be broken but will instead spill outside the text box. Placing a space before this word will create an empty line and push the word onto the next line.

POINT

- All POINT entities have the same style defined by the HEADER variable $PDMODE.
- POINT entities can be drawn relative to the view scale or in absolute units.

6.14 Developer Guides

Information about *ezdxf* internals.

6.14.1 Source Code Formatting

Reformat code by Black with the default setting of 88 characters per line:

```
C:\> black <python-file>
```

6.14.2 Type Annotations

The use of type annotations is encouraged. New modules should pass mypy without errors in non-strict mode. Using `# type: ignore` is fine in tricky situations - type annotations should be helpful in understanding the code and not be a burden.

The following global options are required to pass mypy without error messages:

```
[mypy]
python_version = 3.7
ignore_missing_imports = True
```

Read this to learn where mypy searches for config files.

Use the mypy command line option `--ignore-missing-imports` and `-p` to check the whole package from any location in the file system:

```
PS D:\Source\ezdxf.git> mypy --ignore-missing-imports -p ezdxf
Success: no issues found in 255 source files
```

6.14.3 Design

The *Package Design for Developers* section shows the structure of the *ezdxf* package for developers with more experience, which want to have more insight into the package an maybe want to develop add-ons or want contribute to the *ezdxf* package.

!!! UNDER CONSTRUCTION !!!

Package Design for Developers

A DXF document is divided into several sections, this sections are managed by the `Drawing` object. For each section exist a corresponding attribute in the `Drawing` object:

Section	Attribute
HEADER	`Drawing.header`
CLASSES	`Drawing.classes`
TABLES	`Drawing.tables`
BLOCKS	`Drawing.blocks`
ENTITIES	`Drawing.entities`
OBJECTS	`Drawing.objects`

Resource entities (LAYER, STYLE, LTYPE, …) are stored in tables in the TABLES section. A table owns the table entries, the owner handle of table entry is the handle of the table. Each table has a shortcut in the `Drawing` object:

Table	Attribute
APPID	`Drawing.appids`
BLOCK_RECORD	`Drawing.block_records`
DIMSTYLE	`Drawing.dimstyles`
LAYER	`Drawing.layers`
LTYPE	`Drawing.linetypes`
STYLE	`Drawing.styles`
UCS	`Drawing.ucs`
VIEW	`Drawing.views`
VPORT	`Drawing.viewports`

Graphical entities are stored in layouts: *`Modelspace`*, *`Paperspace`* layouts and *`BlockLayout`*. The core management object of this layouts is the BLOCK_RECORD entity (*`BlockRecord`*), the BLOCK_RECORD is the real owner of the entities, the owner handle of the entities is the handle of the BLOCK_RECORD and the BLOCK_RECORD also owns and manages the entity space of the layout which contains all entities of the layout.

For more information about layouts see also: *Layout Management Structures*

For more information about blocks see also: *Block Management Structures*

Non-graphical entities (objects) are stored in the OBJECTS section. Every object has a parent object in the OBJECTS section, most likely a DICTIONARY object, and is stored in the entity space of the OBJECTS section.

For more information about the OBJECTS section see also: *OBJECTS Section*

All table entries, DXF entities and DXF objects are stored in the entities database accessible as `Drawing.entitydb`. The entity database is a simple key, value storage, key is the entity handle, value is the DXF object.

For more information about the DXF data model see also: *Data Model*

Terminology

States

DXF entities and objects can have different states:

UNBOUND
: Entity is not stored in the `Drawing` entity database and DXF attribute `handle` is `None` and attribute `doc` can be `None`

BOUND
: Entity is stored in the `Drawing` entity database, attribute `doc` has a reference to `Drawing` and DXF attribute `handle` is not `None`

UNLINKED
: Entity is not linked to a layout/owner, DXF attribute `owner` is `None`

LINKED
: Entity is linked to a layout/owner, DXF attribute `owner` is not `None`

Virtual Entity
: State: UNBOUND & UNLINKED

Unlinked Entity
: State: BOUND & UNLINKED

Bound Entity
State: BOUND & LINKED

Actions

NEW
Create a new DXF document

LOAD
Load a DXF document from an external source

CREATE
Create DXF structures from NEW or LOAD data

DESTROY
Delete DXF structures

BIND
Bind an entity to a `Drawing`, set entity state to BOUND & UNLINKED and check or create required resources

UNBIND
unbind …

LINK
Link an entity to an owner/layout. This makes an entity to a real DXF entity, which will be exported at the saving process. Any DXF entity can only be linked to **one** parent entity like DICTIONARY or BLOCK_RECORD.

UNLINK
unlink …

Loading a DXF Document

Loading a DXF document from an external source, creates a new `Drawing` object. This loading process has two stages:

First Loading Stage

- LOAD content from external source as `SectionDict`: `loader.load_dxf_structure()`
- LOAD tag structures as `DXFEntity` objects: `loader.load_dxf_entities()`
- BIND entities: `loader.load_and_bind_dxf_content()`; Special handling of the BIND process, because the `Drawing` is not full initialized, a complete validation is not possible at this stage.

Second Loading Stage

Parse `SectionDict`:

- CREATE sections: HEADER, CLASSES, TABLES, BLOCKS and OBJECTS
- CREATE layouts: Blocks, Layouts
- LINK entities to a owner/layout

The ENTITIES section is a relict from older DXF versions and has to be exported including the modelspace and active paperspace entities, but all entities reside in a BLOCK definition, even modelspace and paperspace layouts are only BLOCK definitions and ezdxf has no explicit ENTITIES section.

Source Code: as developer start your journey at `ezdxf.document.Drawing.read()`, which has no public documentation, because package-user should use *`ezdxf.read()`* and *`ezdxf.readfile()`*.

New DXF Document

Creating New DXF Entities

The default constructor of each entity type creates a new virtual entity:

- DXF attribute *owner* is `None`
- DXF attribute *handle* is `None`
- Attribute *doc* is `None`

The `DXFEntity.new()` constructor creates entities with given *owner*, *handle* and *doc* attributes, if *doc* is not `None` and entity is not already bound to a document, the `new()` constructor automatically bind the entity to the given document *doc*.

There exist only two scenarios:

1. UNBOUND: *doc* is `None` and *handle* is `None`
2. BOUND: *doc* is not `None` and *handle* is not `None`

Factory functions

- `new()`, create a new virtual DXF object/entity
- `load()`, load (create) virtual DXF object/entity from DXF tags
- `bind()`, bind an entity to a document, create required resources if necessary (e.g. ImageDefReactor, SEQEND) and raise exceptions for non-existing resources.
  - Bind entity loaded from an external source to a document, all referenced resources must exist, but try to repair as many flaws as possible because errors were created by another application and are not the responsibility of the package-user.
  - Bind an entity from another DXF document, all invalid resources will be removed silently or created (e.g. SEQEND). This is a simple import from another document without resource import, for a more advanced import including resources exist the `importer` add-on.
  - Bootstrap problem for binding loaded table entries and objects in the OBJECTS section! Can't use `Auditor` to repair this objects, because the DXF document is not fully initialized.
- `is_bound()` returns True if *entity* is bound to document *doc*
- `unbind()` function to remove an entity from a document and set state to a virtual entity, which should also *UNLINK* the entity from layout, because an layout can not store a virtual entity.
- `cls()`, returns the class
- `register_entity()`, registration decorator
- `replace_entity()`, registration decorator

Class Interfaces

DXF Entities

- NEW constructor to create an entity from scratch
- LOAD constructor to create an entity loaded from an external source
- DESTROY interface to kill an entity, set entity state to *dead*, which means `entity.is_alive` returns False. All entity iterators like `EntitySpace`, `EntityQuery`, and `EntityDB` must filter (ignore) *dead* entities. Calling `DXFEntity.destroy()` is a regular way to delete entities.
- LINK an entity to a layout by `BlockRecord.link()`, which set the *owner* handle to BLOCK_RECORD handle (= layout key) and add the entity to the entity space of the BLOCK_RECORD and set/clear the paperspace flag.

DXF Objects

- NEW, LOAD, DESTROY see DXF entities
- LINK: Linking an DXF object means adding the entity to a parent object in the OBJECTS section, most likely a DICTIONARY object, and adding the object to the entity space of the OBJECTS section, the root-dict is the only entity in the OBJECTS section which has an invalid owner handle "0". Any other object with an invalid or destroyed owner is an orphaned entity. The audit process destroys and removes orphaned objects.
- Extension dictionaries (ACAD_XDICTIONARY) are DICTIONARY objects located in the OBJECTS sections and can reference/own other entities of the OBJECTS section.
- The root-dictionary is the only entity in the OBJECTS section which has an invalid owner handle "0". Any other object with an invalid or destroyed owner is an orphaned entity.

Layouts

- LINK interface to link an entity to a layout
- UNLINK interface to remove an entity from a layout

Database

- BIND interface to add an entity to the database of a document
- `delete_entity()` interface, same as UNBIND and DESTROY an entity

6.14.4 Internal Data Structures

Entity Database

The *EntityDB* is a simple key/value database to store *DXFEntity* objects by it's handle, every *Drawing* has its own *EntityDB*, stored in the `Drawing` attribute `entitydb`.

Every DXF entity/object, except tables and sections, are represented as `DXFEntity` or inherited types, this entities are stored in the *EntityDB*, database-key is the `dxf.handle` as plain hex string.

All iterators like `keys()`, `values()`, `items()` and `__iter__()` do not yield destroyed entities.

Warning: The `get()` method and the index operator `[]`, return destroyed entities and entities from the trashcan.

class ezdxf.entitydb.**EntityDB**

__getitem__(*handle: str*) → *DXFEntity*

Get entity by *handle*, does not filter destroyed entities nor entities in the trashcan.

__setitem__(*handle: str*, *entity:* DXFEntity) → None

Set *entity* for *handle*.

__delitem__(*handle: str*) → None

Delete entity by *handle*. Removes entity only from database, does not destroy the entity.

__contains__(*item: str* | DXFEntity) → bool

`True` if database contains *handle*.

__len__() → int

Count of database items.

__iter__() → Iterator[str]

Iterable of all handles, does filter destroyed entities but not entities in the trashcan.

get(*handle: str*) → *DXFEntity* | None

Returns entity for *handle* or `None` if no entry exist, does not filter destroyed entities.

next_handle() → str

Returns next unique handle.

keys() → Iterable[str]

Iterable of all handles, does filter destroyed entities.

values() → Iterable[*DXFEntity*]

Iterable of all entities, does filter destroyed entities.

items() → Iterable[Tuple[str, *DXFEntity*]]

Iterable of all (handle, entities) pairs, does filter destroyed entities.

add(*entity:* DXFEntity) → None

Add *entity* to database, assigns a new handle to the *entity* if `entity.dxf.handle` is `None`. Adding the same entity multiple times is possible and creates only a single database entry.

new_trashcan() → Trashcan

Returns a new trashcan, empty trashcan manually by: : func:*Trashcan.clear()*.

trashcan() → Trashcan

Returns a new trashcan in context manager mode, trashcan will be emptied when leaving context.

purge() → None

Remove all destroyed entities from database, but does not empty the trashcan.

query(*query: str = '*'*) → *EntityQuery*

Entity query over all entities in the DXF document.

Parameters

query – query string

See also:

Entity Query String and *Retrieve entities by query language*

Entity Space

class ezdxf.entitydb.**EntitySpace**(*entities: Iterable[DXFEntity] | None = None*)

An *EntitySpace* is a collection of *DXFEntity* objects, that stores only references to DXFEntity objects.

The *Modelspace*, any *Paperspace* layout and *BlockLayout* objects have an *EntitySpace* container to store their entities.

__iter__() → Iterable[*DXFEntity*]

Iterable of all entities, filters destroyed entities.

__getitem__(*index*) → *DXFEntity*

Get entity at index *item*

EntitySpace has a standard Python list like interface, therefore *index* can be any valid list indexing or slicing term, like a single index `layout[-1]` to get the last entity, or an index slice `layout[:10]` to get the first 10 or fewer entities as `list[DXFEntity]`. Does not filter destroyed entities.

__len__() → int

Count of entities including destroyed entities.

has_handle(*handle: str*) → bool

`True` if *handle* is present, does filter destroyed entities.

purge()

Remove all destroyed entities from entity space.

add(*entity:* DXFEntity) → None

Add *entity*.

extend(*entities: Iterable[*DXFEntity*]*) → None

Add multiple *entities*.

remove(*entity:* DXFEntity) → None

Remove *entity*.

clear() → None

Remove all entities.

DXF Types

Required DXF tag interface:

- property `code`: group code as int
- property `value`: tag value of unspecific type
- `dxfstr()`: returns the DXF string
- `clone()`: returns a deep copy of tag

DXFTag Factory Functions

ezdxf.lldxf.types.**dxftag**(*code: int*, *value: Any*) → *DXFTag*

DXF tag factory function.

Parameters

- **code** – group code
- **value** – tag value

Returns: *DXFTag* or inherited

ezdxf.lldxf.types.**tuples_to_tags**(*iterable: Iterable[tuple[int, Any]]*) → Iterable[*DXFTag*]

Returns an iterable if *DXFTag* or inherited, accepts an iterable of (code, value) tuples as input.

DXFTag

class ezdxf.lldxf.types.**DXFTag**(*code: int*, *value: Any*)

Immutable DXFTag class.

Parameters

- **code** – group code as int
- **value** – tag value, type depends on group code

code

group code as int (do not change)

value

tag value (read-only property)

__eq__(*other*) → bool

`True` if *other* and *self* has same content for *code* and *value*.

__getitem__(*index: int*)

Returns *code* for index 0 and *value* for index 1, emulates a tuple.

__hash__()

Hash support, *DXFTag* can be used in sets and as dict key.

__iter__()

Returns (code, value) tuples.

__repr__() → str

Returns representation string `'DXFTag(code, value)'`.

__str__() → str

Returns content string `'(code, value)'`.

clone() → *DXFTag*

Returns a clone of itself, this method is necessary for the more complex (and not immutable) DXF tag types.

dxfstr() → str

Returns the DXF string e.g. `'  0\nLINE\n'`

DXFBinaryTag

class ezdxf.lldxf.types.**DXFBinaryTag**(*DXFTag*)

Immutable BinaryTags class - immutable by design, not by implementation.

dxfstr() → str

Returns the DXF string for all vertex components.

tostring() → str

Returns binary value as single hex-string.

DXFVertex

class ezdxf.lldxf.types.**DXFVertex**(*DXFTag*)

Represents a 2D or 3D vertex, stores only the group code of the x-component of the vertex, because the y-group-code is x-group-code + 10 and z-group-code id x-group-code+20, this is a rule that ALWAYS applies. This tag is *immutable* by design, not by implementation.

Parameters

- **code** – group code of x-component
- **value** – sequence of x, y and optional z values

dxfstr() → str

Returns the DXF string for all vertex components.

dxftags() → Iterable[*DXFTag*]

Returns all vertex components as single *DXFTag* objects.

NONE_TAG

ezdxf.lldxf.types.**NONE_TAG**

Special tag representing a none existing tag.

Tags

A list of *DXFTag*, inherits from Python standard list. Unlike the statement in the DXF Reference "Do not write programs that rely on the order given here", tag order is sometimes essential and some group codes may appear multiples times in one entity. At the worst case (`Material`: normal map shares group codes with diffuse map) using same group codes with different meanings.

class ezdxf.lldxf.tags.**Tags**

Subclass of `list`.

Collection of *DXFTag* as flat list. Low level tag container, only required for advanced stuff.

classmethod from_text(*text: str*) → *Tags*

Constructor from DXF string.

dxftype() → str

Returns DXF type of entity, e.g. `'LINE'`.

get_handle() → str

Get DXF handle. Raises `DXFValueError` if handle not exist.

Returns
handle as plain hex string like `'FF00'`

Raises
DXFValueError – no handle found

replace_handle(*new_handle: str*) → None

Replace existing handle.

Parameters
new_handle – new handle as plain hex string e.g. `'FF00'`

has_tag(*code: int*) → bool

Returns `True` if a *DXFTag* with given group *code* is present.

Parameters
code – group code as int

has_embedded_objects() → bool

get_first_tag(*code: int*, *default=DXFValueError*) → *DXFTag*

Returns first *DXFTag* with given group code or *default*, if *default* != `DXFValueError`, else raises `DXFValueError`.

Parameters

- **code** – group code as int
- **default** – return value for default case or raises `DXFValueError`

get_first_value(*code: int*, *default=DXFValueError*) → Any

Returns value of first *DXFTag* with given group code or default if *default* != `DXFValueError`, else raises `DXFValueError`.

Parameters

- **code** – group code as int
- **default** – return value for default case or raises `DXFValueError`

find_all(*code: int*) → List[*DXFTag*]

Returns a list of *DXFTag* with given group code.

Parameters
code – group code as int

filter(*codes: Iterable[int]*) → Iterable[*DXFTag*]

Iterate and filter tags by group *codes*.

Parameters
codes – group codes to filter

collect_consecutive_tags(*codes: Iterable[int]*, *start: int = 0*, *end: int = None*) → *Tags*

Collect all consecutive tags with group code in *codes*, *start* and *end* delimits the search range. A tag code not in codes ends the process.

Parameters

- **codes** – iterable of group codes

- **start** – start index as int
- **end** – end index as int, `None` for end index = `len(self)`

Returns
collected tags as *Tags*

tag_index (*code: int*, *start: int = 0*, *end: int | None = None*) → int

Return index of first *DXFTag* with given group code.

Parameters

- **code** – group code as int
- **start** – start index as int
- **end** – end index as int, `None` for end index = `len(self)`

update (*tag:* DXFTag)

Update first existing tag with same group code as *tag*, raises `DXFValueError` if tag not exist.

set_first (*tag:* DXFTag)

Update first existing tag with group code `tag.code` or append tag.

remove_tags (*codes: Iterable[int]*) → None

Remove all tags inplace with group codes specified in *codes*.

Parameters
codes – iterable of group codes as int

remove_tags_except (*codes: Iterable[int]*) → None

Remove all tags inplace except those with group codes specified in *codes*.

Parameters
codes – iterable of group codes

pop_tags (*codes: Iterable[int]*) → Iterable[*DXFTag*]

Pop tags with group codes specified in *codes*.

Parameters
codes – iterable of group codes

classmethod strip (*tags:* Tags, *codes: Iterable[int]*) → *Tags*

Constructor from *tags*, strips all tags with group codes in *codes* from tags.

Parameters

- **tags** – iterable of *DXFTag*
- **codes** – iterable of group codes as int

ezdxf.lldxf.tags.**group_tags** (*tags: Iterable[*DXFTag*]*, *splitcode: int = 0*) → Iterable[*Tags*]

Group of tags starts with a SplitTag and ends before the next SplitTag. A SplitTag is a tag with code == splitcode, like (0, 'SECTION') for splitcode == 0.

Parameters

- **tags** – iterable of `DXFTag`
- **splitcode** – group code of split tag

class ezdxf.lldxf.extendedtags.**ExtendedTags**(*tags: Iterable[DXFTag] = None*, *legacy=False*)

Represents the extended DXF tag structure introduced with DXF R13.

Args:
tags: iterable of *DXFTag* legacy: flag for DXF R12 tags

appdata

Application defined data as list of `Tags`

subclasses

Subclasses as list of `Tags`

xdata

XDATA as list of `Tags`

embedded_objects

embedded objects as list of `Tags`

noclass

Short cut to access first subclass.

get_handle() → str

Returns handle as hex string.

dxftype() → str

Returns DXF type as string like "LINE".

replace_handle(*handle: str*) → None

Replace the existing entity handle by a new value.

legacy_repair()

Legacy (DXF R12) tags handling and repair.

clone() → *ExtendedTags*

Shallow copy.

flatten_subclasses()

Flatten subclasses in legacy mode (DXF R12).

There exists DXF R12 with subclass markers, technical incorrect but works if the reader ignore subclass marker tags, unfortunately ezdxf tries to use this subclass markers and therefore R12 parsing by ezdxf does not work without removing these subclass markers.

This method removes all subclass markers and flattens all subclasses into ExtendedTags.noclass.

get_subclass(*name: str*, *pos: int = 0*) → *Tags*

Get subclass *name*.

Parameters

- **name** – subclass name as string like "AcDbEntity"
- **pos** – start searching at subclass *pos*.

has_xdata(*appid: str*) → bool

`True` if has XDATA for *appid*.

get_xdata(*appid: str*) → *Tags*

Returns XDATA for *appid* as `Tags`.

set_xdata (*appid: str*, *tags: IterableTags*) → None

Set *tags* as XDATA for *appid*.

new_xdata (*appid: str*, *tags: 'IterableTags' = None*) → *Tags*

Append a new XDATA block.

Assumes that no XDATA block with the same *appid* already exist:

```
try:
    xdata = tags.get_xdata('EZDXF')
except ValueError:
    xdata = tags.new_xdata('EZDXF')
```

has_app_data (*appid: str*) → bool

`True` if has application defined data for *appid*.

get_app_data (*appid: str*) → *Tags*

Returns application defined data for *appid* as `Tags` including marker tags.

get_app_data_content (*appid: str*) → *Tags*

Returns application defined data for *appid* as `Tags` without first and last marker tag.

set_app_data_content (*appid: str*, *tags: IterableTags*) → None

Set application defined data for *appid* for already exiting data.

new_app_data (*appid: str*, *tags: 'IterableTags' = None*, *subclass_name: str = None*) → *Tags*

Append a new application defined data to subclass *subclass_name*.

Assumes that no app data block with the same *appid* already exist:

```
try:
    app_data = tags.get_app_data('{ACAD_REACTORS', tags)
except ValueError:
    app_data = tags.new_app_data('{ACAD_REACTORS', tags)
```

classmethod from_text (*text: str*, *legacy: bool = False*) → *ExtendedTags*

Create *ExtendedTags* from DXF text.

Packed DXF Tags

Store DXF tags in compact data structures as `list` or `array.array` to reduce memory usage.

class ezdxf.lldxf.packedtags.**TagList** (*data: Iterable = None*)

Store data in a standard Python `list`.

Args:
data: iterable of DXF tag values.

values

Data storage as `list`.

clone () → *TagList*

Returns a deep copy.

classmethod from_tags (*tags:* Tags, *code: int*) → *TagList*

Setup list from iterable tags.

Parameters

- **tags** – tag collection as *Tags*
- **code** – group code to collect

clear() → None

Delete all data values.

class ezdxf.lldxf.packedtags.**TagArray**(*data: Iterable = None*)

TagArray is a subclass of *TagList*, which store data in an `array.array`. Array type is defined by class variable `DTYPE`.

Args:

data: iterable of DXF tag values.

DTYPE

`array.array` type as string

values

Data storage as `array.array`

set_values(*values: Iterable*) → None

Replace data by *values*.

class ezdxf.lldxf.packedtags.**VertexArray**(*data: Iterable = None*)

Store vertices in an `array.array('d')`. Vertex size is defined by class variable *VERTEX_SIZE*.

Args:

data: iterable of vertex values as linear list e.g. `[x1, y1, x2, y2, x3, y3, ...]`.

VERTEX_SIZE

Size of vertex (2 or 3 axis).

__len__() → int

Count of vertices.

__getitem__(*index: int*)

Get vertex at *index*, extended slicing supported.

__setitem__(*index: int*, *point: Sequence[float]*) → None

Set vertex *point* at *index*, extended slicing not supported.

__delitem__(*index: int*) → None

Delete vertex at *index*, extended slicing supported.

__iter__() → Iterator[Sequence[float]]

Returns iterable of vertices.

__str__() → str

String representation.

insert(*pos: int*, *point: Sequence[float]*)

Insert *point* in front of vertex at index *pos*.

Parameters

- **pos** – insert position
- **point** – point as tuple

append(*point: Sequence[float]*) → None

Append *point*.

extend (*points: Iterable[Sequence[float]]*) → None

Extend array by *points*.

set (*points: Iterable[Sequence[float]]*) → None

Replace all vertices by *points*.

clear () → None

Delete all vertices.

clone () → *VertexArray*

Returns a deep copy.

classmethod from_tags (*tags: Iterable[*DXFTag*]*, *code: int = 10*) → *VertexArray*

Setup point array from iterable tags.

Parameters

- **tags** – iterable of *DXFVertex*
- **code** – group code to collect

export_dxf (*tagwriter: AbstractTagWriter*, *code=10*)

XData

class ezdxf.entities.xdata.**XData**

Internal management class for XDATA.

See also:

- XDATA user reference: *Extended Data (XDATA)*
- Wrapper class to store a list in XDATA: *XDataUserList*
- Wrapper class to store a dict in XDATA: *XDataUserDict*
- Tutorial: *Storing Custom Data in DXF Files*
- DXF Internals: *Extended Data*
- DXF R2018 Reference

__contains__ (*appid: str*) → bool

Returns `True` if DXF tags for *appid* exist.

add (*appid: str*, *tags: Iterable[tuple[int, Any]* | DXFTag*]*) → None

Add a list of DXF tags for *appid*. The *tags* argument is an iterable of (group code, value) tuples, where the group code has to be an integer value. The mandatory XDATA marker (1001, appid) is added automatically if front of the tags if missing.

Each entity can contain only one list of tags for each *appid*. Adding a second list of tags for the same *appid* replaces the existing list of tags.

The valid XDATA group codes are restricted to some specific values in the range from 1000 to 1071, for more information see also the internals about *Extended Data*.

get (*appid: str*) → *Tags*

Returns the DXF tags as *Tags* list stored by *appid*.

Raises
DXFValueError – no data for *appid* exist

discard(*appid*)

Delete DXF tags for *appid*. None existing appids are silently ignored.

has_xlist(*appid: str*, *name: str*) → bool

Returns `True` if list *name* from XDATA *appid* exists.

Parameters

- **appid** – APPID
- **name** – list name

get_xlist(*appid: str*, *name: str*) → list[tuple]

Get list *name* from XDATA *appid*.

Parameters

- **appid** – APPID
- **name** – list name

Returns: list of DXFTags including list name and curly braces '{' '}' tags

Raises

- ***DXFKeyError*** – XDATA *appid* does not exist
- ***DXFValueError*** – list *name* does not exist

set_xlist(*appid: str*, *name: str*, *tags: Iterable*) → None

Create new list *name* of XDATA *appid* with *xdata_tags* and replaces list *name* if already exists.

Parameters

- **appid** – APPID
- **name** – list name
- **tags** – list content as DXFTags or (code, value) tuples, list name and curly braces '{' '}' tags will be added

discard_xlist(*appid: str*, *name: str*) → None

Deletes list *name* from XDATA *appid*. Ignores silently if XDATA *appid* or list *name* not exist.

Parameters

- **appid** – APPID
- **name** – list name

replace_xlist(*appid: str*, *name: str*, *tags: Iterable*) → None

Replaces list *name* of existing XDATA *appid* by *tags*. Appends new list if list *name* do not exist, but raises *DXFValueError* if XDATA *appid* do not exist.

Low level interface, if not sure use *set_xdata_list()* instead.

Parameters

- **appid** – APPID
- **name** – list name
- **tags** – list content as DXFTags or (code, value) tuples, list name and curly braces '{' '}' tags will be added

Raises
DXFValueError – XDATA *appid* do not exist

transform (*m:* Matrix44) → None

Transform XDATA tags with group codes 1011, 1012, 1013, 1041 and 1042 inplace. For more information see *Extended Data* Internals.

Application-Defined Data (AppData)

Starting at DXF R13, DXF objects can contain application-defined codes (AppData) outside of XDATA.

All AppData is defined with a beginning (102, "{APPID") tag and according to the DXF reference appear should appear before the first subclass marker.

There are two known use cases of this data structure in Autodesk products:

- `ACAD_REACTORS`, store handles to persistent reactors in a DXF entity
- `ACAD_XDICTIONARY`, store handle to the extension dictionary of a DXF entity

Both AppIDs are not defined/stored in the AppID table!

class ezdxf.entities.appdata.**AppData**

Internal management class for Application defined data.

See also:

- User reference: *Application-Defined Data (AppData)*
- Internals about *Application-Defined Codes* tags

__contains__ (*appid: str*) → bool

Returns `True` if application-defined data exist for *appid*.

__len__ () → int

Returns the count of AppData.

add (*appid: str*, *data: Iterable[Sequence]*) → None

Add application-defined tags for *appid*. Adds first tag (102, "{APPID") if not exist. Adds last tag (102, "}" if not exist.

get (*appid: str*) → *Tags*

Get application-defined data for *appid* as `Tags` container. The first tag is always (102, "{APPID"). The last tag is always (102, "}").

set (*tags:* Tags) → None

Store raw application-defined data tags. The first tag has to be (102, "{APPID"). The last tag has to be (102, "}").

discard (*appid: str*)

Delete application-defined data for *appid* without raising and error if *appid* doesn't exist.

Reactors

class ezdxf.entities.appdata.**Reactors**

Internal management class for persistent reactor handles. Handles are stored as hex strings like `"ABBA"`.

See also:

- User reference: *Reactors*
- Internals about *Persistent Reactors* tags

__contains__(*handle: str*) → bool

Returns `True` if *handle* is registered.

__len__() → int

Returns count of registered handles.

__iter__() → Iterator[str]

Returns an iterator for all registered handles.

add(*handle: str*) → None

Add a single *handle*.

get() → list[str]

Returns all registered handles as sorted list.

set(*handles: Iterable[str] | None*) → None

Reset all handles.

discard(*handle: str*)

Discard a single *handle*.

6.14.5 Documentation Guide

Formatting Guide

This section is only for myself, because of the long pauses between develop iterations, I often forget to be consistent in documentation formatting.

Documentation is written with Sphinx and reSturcturedText.

Started integration of documentation into source code and using autodoc features of Sphinx wherever useful.

Sphinx theme provided by Read the Docs :

```
pip install sphinx-rtd-theme
```

guide — Example module

guide.**example_func**(*a: int*, *b: str*, *test: str = None*, *flag: bool = True*) → None

Parameters *a* and *b* are positional arguments, argument *test* defaults to `None` and *flag* to `True`. Set *a* to 70 and *b* to "x" as an example. Inline code examples `example_func(70, 'x')` or simple `example_func(70, "x")`

- arguments: *a*, *b*, *test* and *flags*
- literal number values: 1, 2 … 999
- literal string values: "a String"
- literal tags: (5, "F000")
- inline code: call a `example_func(x)`
- Python keywords: `None`, `True`, `False`, `tuple`, `list`, `dict`, `str`, `int`, `float`
- Exception classes: `DXFAttributeError`

class guide.**ExampleCls**(***kwargs*)

The *ExampleCls* constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
e = ExampleCls(flag=True)
```

flag

This is the attribute *flag*.

set_axis(*axis*)

axis as (x, y, z) tuple

Args:
axis: (x, y, z) tuple

example_method(*flag: bool = False*) → None

Method *example_method()* of class *ExampleCls*

Text Formatting

DXF version
DXF R12 (AC1009), DXF R2004 (AC1018)

DXF Types
DXF types are always written in uppercase letters but without further formatting: DXF, LINE, CIRCLE

(internal API)
Marks methods as internal API, gets no public documentation.

(internal class)
Marks classes only for internal usage, gets not public documentation.

Spatial Dimensions
2D and 3D with an uppercase letter D

Axis
x-axis, y-axis and z-axis

Planes
xy-plane, xz-plane, yz-plane

Layouts
modelspace, paperspace [layout], block [layout]

Extended Entity Data
AppData, XDATA, embedded object, APPID

6.15 Glossary

ACI
AutoCAD Color Index (ACI)

ACIS
The 3D ACIS Modeler (ACIS) is a geometric modeling kernel developed by Spatial Corp. ® (formerly *Spatial Technology*) and now part of *Dassault Systems*. All ACIS based DXF entities store their geometry as *SAT* or *SAB* data. These are not open data formats and a license has to be purchased to get access to their SDK, therefore *ezdxf* can not provide any support for creating, processing or transforming of ACIS based DXF entities.

bulge
The *Bulge value* is used to create arc shaped line segments in `Polyline` and `LWPolyline` entities.

CAD
Computer-Assisted Drafting or Computer-Aided Design

CTB
Color dependent plot style table (`ColorDependentPlotStyles`)

DWG
Proprietary file format of AutoCAD ®. Documentation for this format is available from the Open Design Alliance (ODA) at their Downloads section. This documentation is created by reverse engineering therefore not perfect nor complete.

DXF
Drawing eXchange Format is a file format used by AutoCAD ® to interchange data with other *CAD* applications. DXF is a trademark of Autodesk ®. See also *What is DXF?*

proxy-graphic
The proxy-graphic is an internal data format to add a graphical representation to DXF entities which are unknown (custom DXF entities), not documented or very complex so CAD applications can display them without knowledge about the internal structure of these entities.

raw-color
Raw color value as stored in DWG files, this integer value can represent *ACI* values as well as and *true-color* values

reliable CAD application
CAD applications which create valid DXF documents in the meaning and interpretation of Autodesk. See also *What is DXF?*

SAB
ACIS file format (Standard ACIS Binary), binary stored data

SAT
ACIS file format (Standard ACIS Text), data stored as ASCII text

STB
Named plot style table (`NamedPlotStyles`)

true-color
RGB color representation, a combination red, green and blue values to define a color.

6.16 Indices and tables

- genindex
- search

PYTHON MODULE INDEX

e

g

INDEX

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z