cattrs Documentation

Release 23.1.2

Tin Tvrtković

Aug 28, 2023

CONTENTS

All *cattrs* functionality is exposed through a `cattrs.Converter` object. Global *cattrs* functions, such as `cattrs.unstructure()`, use a single global converter. Changes done to this global converter, such as registering new structure and unstructure hooks, affect all code using the global functions.

CHAPTER

ONE

GLOBAL CONVERTER

A global converter is provided for convenience as `cattrs.global_converter`. The following functions implicitly use this global converter:

- `cattrs.structure()`
- `cattrs.unstructure()`
- `cattrs.structure_attrs_fromtuple()`
- `cattrs.structure_attrs_fromdict()`

Changes made to the global converter will affect the behavior of these functions.

Larger applications are strongly encouraged to create and customize a different, private instance of `cattrs.Converter`.

CHAPTER

TWO

CONVERTER OBJECTS

To create a private converter, simply instantiate a `cattrs.Converter`. Currently, a converter contains the following state:

- a registry of unstructure hooks, backed by a singledispatch and a `function_dispatch`.
- a registry of structure hooks, backed by a different singledispatch and `function_dispatch`.
- a LRU cache of union disambiguation functions.
- a reference to an unstructuring strategy (either AS_DICT or AS_TUPLE).
- a `dict_factory` callable, used for creating `dicts` when dumping *attrs* classes using `AS_DICT`.

Converters may be cloned using the `cattrs.Converter.copy()` method. The new copy may be changed through the `copy` arguments, but will retain all manually registered hooks from the original.

CHAPTER

THREE

CATTRS.CONVERTER

The `Converter` is a converter variant that automatically generates, compiles and caches specialized structuring and unstructuring hooks for *attrs* classes and dataclasses.

`Converter` differs from the `cattrs.BaseConverter` in the following ways:

- structuring and unstructuring of *attrs* classes is slower the first time, but faster every subsequent time
- structuring and unstructuring can be customized
- support for *attrs* classes with PEP563 (postponed) annotations
- support for generic *attrs* classes
- support for easy overriding collection unstructuring

The `Converter` used to be called `GenConverter`, and that alias is still present for backwards compatibility reasons.

CHAPTER

FOUR

CATTRS.BASECONVERTER

The `BaseConverter` is a simpler and slower Converter variant. It does no code generation, so it may be faster on first-use which can be useful in specific cases, like CLI applications where startup time is more important than throughput.

CHAPTER

FIVE

COMMON USAGE EXAMPLES

This section covers common use examples of *cattrs* features.

5.1 Using Pendulum for Dates and Time

To use the excellent Pendulum library for datetimes, we need to register structuring and unstructuring hooks for it.

First, we need to decide on the unstructured representation of a datetime instance. Since all our datetimes will use the UTC time zone, we decide to use the UNIX epoch timestamp as our unstructured representation.

Define a class using Pendulum's `DateTime`:

```
>>> import pendulum
>>> from pendulum import DateTime

>>> @define
... class MyRecord:
...     a_string: str
...     a_datetime: DateTime
```

Next, we register hooks for the `DateTime` class on a new `Converter` instance.

```
>>> from cattrs import Converter

>>> converter = Converter()

>>> converter.register_unstructure_hook(DateTime, lambda dt: dt.timestamp())
>>> converter.register_structure_hook(DateTime, lambda ts, _: pendulum.from_
↪timestamp(ts))
```

And we can proceed with unstructuring and structuring instances of `MyRecord`.

```
>>> my_record = MyRecord('test', pendulum.datetime(2018, 7, 28, 18, 24))
>>> my_record
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'UTC')))

>>> converter.unstructure(my_record)
{'a_string': 'test', 'a_datetime': 1532802240.0}

>>> converter.structure({'a_string': 'test', 'a_datetime': 1532802240.0}, MyRecord)
```

(continues on next page)

(continued from previous page)

```
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'UTC')))
```

After a while, we realize we *will* need our datetimes to have timezone information. We decide to switch to using the ISO 8601 format for our unstructured datetime instances.

```
>>> converter = cattrs.Converter()
>>> converter.register_unstructure_hook(DateTime, lambda dt: dt.to_iso8601_string())
>>> converter.register_structure_hook(DateTime, lambda isostring, _: pendulum.
↪parse(isostring))

>>> my_record = MyRecord('test', pendulum.datetime(2018, 7, 28, 18, 24, tz='Europe/Paris
↪'))
>>> my_record
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'Europe/Paris')))

>>> converter.unstructure(my_record)
{'a_string': 'test', 'a_datetime': '2018-07-28T18:24:00+02:00'}

>>> converter.structure({'a_string': 'test', 'a_datetime': '2018-07-28T18:24:00+02:00'},␣
↪MyRecord)
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'+02:00')))
```

5.2 Using factory hooks

For this example, let's assume you have some attrs classes with snake case attributes, and you want to un/structure them as camel case.

Warning: A simpler and better approach to this problem is to simply make your class attributes camel case. However, this is a good example of the power of hook factories and *cattrs'* component-based design.

Here's our simple data model:

```
@define
class Inner:
    a_snake_case_int: int
    a_snake_case_float: float
    a_snake_case_str: str

@define
class Outer:
    a_snake_case_inner: Inner
```

Let's examine our options one by one, starting with the simplest: writing manual un/structuring hooks.

We just write the code by hand and register it:

```
def unstructure_inner(inner):
    return {
        "aSnakeCaseInt": inner.a_snake_case_int,
        "aSnakeCaseFloat": inner.a_snake_case_float,
        "aSnakeCaseStr": inner.a_snake_case_str
    }

>>> converter.register_unstructure_hook(Inner, unstructure_inner)
```

(Let's skip the other unstructure hook and 2 structure hooks due to verbosity.)

This will get us where we want to go, but the drawbacks are immediately obvious: we'd need to write a ton of code ourselves, wasting effort, increasing our maintenance burden and risking bugs. Obviously this won't do.

Why write code when we can write code to write code for us? In this case this code has already been written for you. *cattrs* contains a module, `cattrs.gen`, with functions to automatically generate hooks exactly like this. These functions also take parameters to customize the generated hooks.

We can generate and register the renaming hooks we need:

```
>>> from cattrs.gen import make_dict_unstructure_fn, override

>>> converter.register_unstructure_hook(
...     Inner,
...      make_dict_unstructure_fn(
...          Inner,
...          converter,
...          a_snake_case_int=override(rename="aSnakeCaseInt"),
...          a_snake_case_float=override(rename="aSnakeCaseFloat"),
...          a_snake_case_str=override(rename="aSnakeCaseStr"),
...      )
...  )
```

(Again skipping the other hooks due to verbosity.)

This is still too verbose and manual for our tastes, so let's automate it further. We need a way to convert snake case identifiers to camel case, so let's grab one from Stack Overflow:

```
def to_camel_case(snake_str: str) -> str:
    components = snake_str.split("_")
    return components[0] + "".join(x.title() for x in components[1:])
```

We can combine this with `attrs.fields` to save us some typing:

```
from attrs import fields
from cattrs.gen import make_dict_unstructure_fn, override

converter.register_unstructure_hook(
    Inner,
    make_dict_unstructure_fn(
        Inner,
        converter,
        **{a.name: override(rename=to_camel_case(a.name)) for a in fields(Inner)}
    )
)
```

(continues on next page)

(continued from previous page)

```
converter.register_unstructure_hook(
    Outer,
    make_dict_unstructure_fn(
        Outer,
        converter,
        **{a.name: override(rename=to_camel_case(a.name)) for a in fields(Outer)}
    )
)
```

(Skipping the structuring hooks due to verbosity.)

Now we're getting somewhere, but we still need to do this for each class separately. The final step is using hook factories instead of hooks directly.

Hook factories are functions that return hooks. They are also registered using predicates instead of being attached to classes directly, like normal un/structure hooks. Predicates are functions that given a type return a boolean whether they handle it.

We want our hook factories to trigger for all *attrs* classes, so we need a predicate to recognize whether a type is an *attrs* class. Luckily, *attrs* comes with `attrs.has`, which is exactly this.

As the final step, we can combine all of this into two hook factories:

```
from attrs import has, fields
from cattrs import Converter
from cattrs.gen import make_dict_unstructure_fn, make_dict_structure_fn, override

converter = Converter()

def to_camel_case(snake_str: str) -> str:
    components = snake_str.split("_")
    return components[0] + "".join(x.title() for x in components[1:])

def to_camel_case_unstructure(cls):
    return make_dict_unstructure_fn(
        cls,
        converter,
        **{
            a.name: override(rename=to_camel_case(a.name))
            for a in fields(cls)
        }
    )

def to_camel_case_structure(cls):
    return make_dict_structure_fn(
        cls,
        converter,
        **{
            a.name: override(rename=to_camel_case(a.name))
            for a in fields(cls)
        }
    )
```

(continues on next page)

(continued from previous page)

```
converter.register_unstructure_hook_factory(
    has, to_camel_case_unstructure
)
converter.register_structure_hook_factory(
    has, to_camel_case_structure
)
```

The `converter` instance will now un/structure every attrs class to camel case. Nothing has been omitted from this final example; it's complete.

5.3 Using fallback key names

Sometimes when structuring data, the input data may be in multiple formats that need to be converted into a common attribute.

Consider an example where a data store creates a new schema version and renames a key (ie, `{'old_field': 'value1'}` in v1 becomes `{'new_field': 'value1'}` in v2), while also leaving existing records in the system with the V1 schema. Both keys should convert to the same field.

Here, builtin customizations such as *rename* are insufficient - *cattrs* cannot structure both `old_field` and `new_field` into a single field using `rename`, at least not on the same converter.

In order to support both fields, you can apply a little preprocessing to the default *cattrs* structuring hooks. One approach is to write the following decorator and apply it to your class.

```
from attrs import define
from cattrs import Converter
from cattrs.gen import make_dict_structure_fn

converter = Converter()


def fallback_field(
    converter_arg: Converter,
    old_to_new_field: dict[str, str]
):
    def decorator(cls):
        struct = make_dict_structure_fn(cls, converter_arg)

        def structure(d, cl):
            for k, v in old_to_new_field.items():
                if k in d:
                    d[v] = d[k]

            return struct(d, cl)

        converter_arg.register_structure_hook(cls, structure)

        return cls

    return decorator
```

(continues on next page)

(continued from previous page)

```
@fallback_field(converter, {"old_field": "new_field"})
@define
class MyInternalAttr:
    new_field: str
```

cattrs will now structure both key names into `new_field` on your class.

```
converter.structure({"new_field": "foo"}, MyInternalAttr)
converter.structure({"old_field": "foo"}, MyInternalAttr)
```

CHAPTER

SIX

WHAT YOU CAN STRUCTURE AND HOW

The philosophy of *cattrs* structuring is simple: give it an instance of Python built-in types and collections, and a type describing the data you want out. *cattrs* will convert the input data into the type you want, or throw an exception.

All structuring conversions are composable, where applicable. This is demonstrated further in the examples.

6.1 Primitive Values

6.1.1 `typing.Any`

Use `typing.Any` to avoid applying any conversions to the object you're structuring; it will simply be passed through.

```
>>> cattrs.structure(1, Any)
1
>>> d = {1: 1}
>>> cattrs.structure(d, Any) is d
True
```

6.1.2 `int`, `float`, `str`, `bytes`

Use any of these primitive types to convert the object to the type.

```
>>> cattrs.structure(1, str)
'1'
>>> cattrs.structure("1", float)
1.0
```

In case the conversion isn't possible, the expected exceptions will be propagated out. The particular exceptions are the same as if you'd tried to do the conversion yourself, directly.

```
>>> cattrs.structure("not-an-int", int)
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'not-an-int'
```

6.1.3 Enums

Enums will be structured by their values. This works even for complex values, like tuples.

```
>>> @unique
... class CatBreed(Enum):
...     SIAMESE = "siamese"
...     MAINE_COON = "maine_coon"
...     SACRED_BIRMAN = "birman"

>>> cattrs.structure("siamese", CatBreed)
<CatBreed.SIAMESE: 'siamese'>
```

Again, in case of errors, the expected exceptions will fly out.

```
>>> cattrs.structure("alsatian", CatBreed)
Traceback (most recent call last):
...
ValueError: 'alsatian' is not a valid CatBreed
```

6.1.4 pathlib.Path

`pathlib.Path` objects are structured using their string value.

```
>>> from pathlib import Path

>>> cattrs.structure("/root", Path)
PosixPath('/root')
```

In case the conversion isn't possible, the resulting exception is propagated out.

New in version 23.1.0.

6.2 Collections and Other Generics

6.2.1 Optionals

`Optional` primitives and collections are supported out of the box.

```
>>> cattrs.structure(None, int)
Traceback (most recent call last):
...
TypeError: int() argument must be a string, a bytes-like object or a number, not
↪'NoneType'
>>> cattrs.structure(None, Optional[int])
>>> # None was returned.
```

Bare `Optional` s (non-parameterized, just `Optional`, as opposed to `Optional[str]`) aren't supported, use `Optional[Any]` instead.

The Python 3.10 more readable syntax, `str | None` instead of `Optional[str]`, is also supported.

This generic type is composable with all other converters.

```
>>> cattrs.structure(1, Optional[float])
1.0
```

6.2.2 Lists

Lists can be produced from any iterable object. Types converting to lists are:

- `Sequence[T]`
- `MutableSequence[T]`
- `List[T]`
- `list[T]`

In all cases, a new list will be returned, so this operation can be used to copy an iterable into a list. A bare type, for example `Sequence` instead of `Sequence[int]`, is equivalent to `Sequence[Any]`.

```
>>> cattrs.structure((1, 2, 3), MutableSequence[int])
[1, 2, 3]
```

These generic types are composable with all other converters.

```
>>> cattrs.structure((1, None, 3), list[Optional[str]])
['1', None, '3']
```

6.2.3 Deques

Deques can be produced from any iterable object. Types converting to deques are:

- `Deque[T]`
- `deque[T]`

In all cases, a new **unbounded** deque (`maxlen=None`) will be returned, so this operation can be used to copy an iterable into a deque. If you want to convert into bounded `deque`, registering a custom structuring hook is a good approach.

```
>>> cattrs.structure((1, 2, 3), deque[int])
deque([1, 2, 3])
```

These generic types are composable with all other converters.

```
>>> cattrs.structure((1, None, 3), deque[Optional[str]])
deque(['1', None, '3'])
```

New in version 23.1.0.

6.2.4 Sets and Frozensets

Sets and frozensets can be produced from any iterable object. Types converting to sets are:

- `Set[T]`
- `MutableSet[T]`
- `set[T]`

Types converting to frozensets are:

- `FrozenSet[T]`
- `frozenset[T]`

In all cases, a new set or frozenset will be returned, so this operation can be used to copy an iterable into a set. A bare type, for example `MutableSet` instead of `MutableSet[int]`, is equivalent to `MutableSet[Any]`.

```
>>> cattrs.structure([1, 2, 3, 4], Set)
{1, 2, 3, 4}
```

These generic types are composable with all other converters.

```
>>> cattrs.structure([[1, 2], [3, 4]], set[frozenset[str]])
{frozenset({'1', '2'}), frozenset({'4', '3'})}
```

6.2.5 Dictionaries

Dicts can be produced from other mapping objects. To be more precise, the object being converted must expose an `items()` method producing an iterable key-value tuples, and be able to be passed to the `dict` constructor as an argument. Types converting to dictionaries are:

- `Dict[K, V]`
- `MutableMapping[K, V]`
- `Mapping[K, V]`
- `dict[K, V]`

In all cases, a new dict will be returned, so this operation can be used to copy a mapping into a dict. Any type parameters set to `typing.Any` will be passed through unconverted. If both type parameters are absent, they will be treated as `Any` too.

```
>>> from collections import OrderedDict
>>> cattrs.structure(OrderedDict([(1, 2), (3, 4)]), Dict)
{1: 2, 3: 4}
```

These generic types are composable with all other converters. Note both keys and values can be converted.

```
>>> cattrs.structure({1: None, 2: 2.0}, dict[str, Optional[int]])
{'1': None, '2': 2}
```

6.2.6 Typed Dicts

TypedDicts can be produced from mapping objects, usually dictionaries.

```
>>> from typing import TypedDict

>>> class MyTypedDict(TypedDict):
...     a: int

>>> cattrs.structure({"a": "1"}, MyTypedDict)
{'a': 1}
```

Both *total* and *non-total* TypedDicts are supported, and inheritance between any combination works (except on 3.8 when `typing.TypedDict` is used, see below). Generic TypedDicts work on Python 3.11 and later, since that is the first Python version that supports them in general.

`typing.Required` and `typing.NotRequired` are supported.

On Python 3.7, using `typing_extensions.TypedDict` is required since `typing.TypedDict` doesn't exist there. On Python 3.8, using `typing_extensions.TypedDict` is recommended since `typing.TypedDict` doesn't support all necessary features, so certain combinations of subclassing, totality and `typing.Required` won't work.

Similar to attrs *classes*, structuring can be customized using `cattrs.gen.typeddicts.make_dict_structure_fn()`.

```
>>> from typing import TypedDict
>>> from cattrs import Converter
>>> from cattrs.gen import override
>>> from cattrs.gen.typeddicts import make_dict_structure_fn

>>> class MyTypedDict(TypedDict):
...     a: int
...     b: int

>>> c = Converter()
>>> c.register_structure_hook(
...     MyTypedDict,
...     make_dict_structure_fn(
...         MyTypedDict,
...         c,
...         a=override(rename="a-with-dash")
...     )
... )

>>> c.structure({"a-with-dash": 1, "b": 2}, MyTypedDict)
{'b': 2, 'a': 1}
```

See also:

Unstructuring TypedDicts.

New in version 23.1.0.

6.2.7 Homogeneous and Heterogeneous Tuples

Homogeneous and heterogeneous tuples can be produced from iterable objects. Heterogeneous tuples require an iterable with the number of elements matching the number of type parameters exactly. Use:

- `Tuple[A, B, C, D]`
- `tuple[A, B, C, D]`

Homogeneous tuples use:

- `Tuple[T, ...]`
- `tuple[T, ...]`

In all cases a tuple will be returned. Any type parameters set to `typing.Any` will be passed through unconverted.

```
>>> cattrs.structure([1, 2, 3], tuple[int, str, float])
(1, '2', 3.0)
```

The tuple conversion is composable with all other converters.

```
>>> cattrs.structure([{1: 1}, {2: 2}], tuple[dict[str, float], ...])
({'1': 1.0}, {'2': 2.0})
```

6.2.8 Unions

Unions of `NoneType` and a single other type are supported (also known as `Optional` s). All other unions require a disambiguation function.

Automatic Disambiguation

In the case of a union consisting exclusively of `attrs` classes, `cattrs` will attempt to generate a disambiguation function automatically; this will succeed only if each class has a unique field. Given the following classes:

```
>>> @define
... class A:
...     a = field()
...     x = field()

>>> @define
... class B:
...     a = field()
...     y = field()

>>> @define
... class C:
...     a = field()
...     z = field()
```

`cattrs` can deduce only instances of `A` will contain `x`, only instances of `B` will contain `y`, etc. A disambiguation function using this information will then be generated and cached. This will happen automatically, the first time an appropriate union is structured.

Manual Disambiguation

To support arbitrary unions, register a custom structuring hook for the union (see *Registering custom structuring hooks*).

Another option is to use a custom tagged union strategy (see *Strategies - Tagged Unions*).

6.2.9 typing.Final

PEP 591 Final attribute types (`Final[int]`) are supported and structured appropriately.

New in version 23.1.0.

See also:

Unstructuring Final.

6.2.10 typing.Annotated

PEP 593 annotations (`typing.Annotated[type, ...]`) are supported and are matched using the first type present in the annotated type.

6.2.11 typing.NewType

NewTypes are supported and are structured according to the rules for their underlying type. Their hooks can also be overriden using :py:attr:`cattrs.Converter.register_structure_hook`.

```
>>> from typing import NewType
>>> from datetime import datetime

>>> IsoDate = NewType("IsoDate", datetime)

>>> converter = cattrs.Converter()
>>> converter.register_structure_hook(IsoDate, lambda v, _: datetime.fromisoformat(v))

>>> converter.structure("2022-01-01", IsoDate)
datetime.datetime(2022, 1, 1, 0, 0)
```

New in version 22.2.0.

See also:

Unstructuring NewTypes.

Note: NewTypes are not supported by the legacy BaseConverter.

6.3 *attrs* Classes and Dataclasses

6.3.1 Simple *attrs* Classes and Dataclasses

attrs classes and dataclasses using primitives, collections of primitives and their own converters work out of the box. Given a mapping `d` and class `A`, *cattrs* will simply instantiate `A` with `d` unpacked.

```
>>> @define
... class A:
...     a: int
...     b: int

>>> cattrs.structure({'a': 1, 'b': '2'}, A)
A(a=1, b=2)
```

Classes like these deconstructed into tuples can be structured using `structure_attrs_fromtuple()` (`fromtuple` as in the opposite of `attr.astuple` and `converter.unstructure_attrs_astuple`).

```
>>> @define
... class A:
...     a: str
...     b: int

>>> cattrs.structure_attrs_fromtuple(['string', '2'], A)
A(a='string', b=2)
```

Loading from tuples can be made the default by creating a new `Converter` with `unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE`.

```
>>> converter = cattrs.Converter(unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE)
>>> @define
... class A:
...     a: str
...     b: int

>>> converter.structure(['string', '2'], A)
A(a='string', b=2)
```

Structuring from tuples can also be made the default for specific classes only; see registering custom structure hooks below.

6.4 Using Attribute Types and Converters

By default, `structure()` will use hooks registered using `register_structure_hook()`, to convert values to the attribute type, and fallback to invoking any converters registered on attributes with `attrib`.

```
>>> from ipaddress import IPv4Address, ip_address
>>> converter = cattrs.Converter()

# Note: register_structure_hook has not been called, so this will fallback to 'ip_address
↪'
```

(continues on next page)

(continued from previous page)

```
>>> @define
... class A:
...     a: IPv4Address = field(converter=ip_address)

>>> converter.structure({'a': '127.0.0.1'}, A)
A(a=IPv4Address('127.0.0.1'))
```

Priority is still given to hooks registered with `register_structure_hook()`, but this priority can be inverted by setting `prefer_attrib_converters` to `True`.

```
>>> converter = cattrs.Converter(prefer_attrib_converters=True)

>>> converter.register_structure_hook(int, lambda v, t: int(v))

>>> @define
... class A:
...     a: int = field(converter=lambda v: int(v) + 5)

>>> converter.structure({'a': '10'}, A)
A(a=15)
```

6.4.1 Complex `attrs` Classes and Dataclasses

Complex `attrs` classes and dataclasses are classes with type information available for some or all attributes. These classes support almost arbitrary nesting.

Type information is supported by attrs directly, and can be set using type annotations when using Python 3.6+, or by passing the appropriate type to `attr.ib`.

```
>>> @define
... class A:
...     a: int

>>> attr.fields(A).a
Attribute(name='a', default=NOTHING, validator=None, repr=True, eq=True, eq_key=None,␣
↪order=True, order_key=None, hash=None, init=True, metadata=mappingproxy({}), type=
↪<class 'int'>, converter=None, kw_only=False, inherited=False, on_setattr=None, alias=
↪'a')
```

Type information, when provided, can be used for all attribute types, not only attributes holding `attrs` classes and dataclasses.

```
>>> @define
... class A:
...     a: int = 0

>>> @define
... class B:
...     b: A

>>> cattrs.structure({'b': {'a': '1'}}, B)
B(b=A(a=1))
```

Finally, if an `attrs` or `dataclass` class uses inheritance and as such has one or several subclasses, it can be structured automatically to its exact subtype by using the *include subclasses* strategy.

6.5 Registering Custom Structuring Hooks

cattrs doesn't know how to structure non-*attrs* classes by default, so it has to be taught. This can be done by registering structuring hooks on a converter instance (including the global converter).

Here's an example involving a simple, classic (i.e. non-*attrs*) Python class.

```
>>> class C:
...     def __init__(self, a):
...         self.a = a
...     def __repr__(self):
...         return f'C(a={self.a})'

>>> cattrs.structure({'a': 1}, C)
Traceback (most recent call last):
...
StructureHandlerNotFoundError: Unsupported type: <class '__main__.C'>. Register a␣
↪structure hook for it.

>>> cattrs.register_structure_hook(C, lambda d, t: C(**d))
>>> cattrs.structure({'a': 1}, C)
C(a=1)
```

The structuring hooks are callables that take two arguments: the object to convert to the desired class and the type to convert to. (The type may seem redundant but is useful when dealing with generic types.)

When using `cattrs.register_structure_hook()`, the hook will be registered on the global converter. If you want to avoid changing the global converter, create an instance of `cattrs.Converter` and register the hook on that.

In some situations, it is not possible to decide on the converter using typing mechanisms alone (such as with *attrs* classes). In these situations, *cattrs* provides a `register_unstructure_hook_func()` hook instead, which accepts a predicate function to determine whether that type can be handled instead.

The function-based hooks are evaluated after the class-based hooks. In the case where both a class-based hook and a function-based hook are present, the class-based hook will be used.

```
>>> class D:
...     custom = True
...     def __init__(self, a):
...         self.a = a
...     def __repr__(self):
...         return f'D(a={self.a})'
...     @classmethod
...     def deserialize(cls, data):
...         return cls(data["a"])

>>> cattrs.register_structure_hook_func(
...     lambda cls: getattr(cls, "custom", False), lambda d, t: t.deserialize(d)
... )
```

(continues on next page)

(continued from previous page)

```
>>> cattrs.structure({'a': 2}, D)
D(a=2)
```

6.6 Structuring Hook Factories

Hook factories operate one level higher than structuring hooks; structuring hooks are functions registered to a class or predicate, and hook factories are functions (registered via a predicate) that produce structuring hooks.

Structuring hooks factories are registered using `Converter.register_structure_hook_factory()`.

Here's a small example showing how to use factory hooks to apply the `forbid_extra_keys` to all attrs classes:

```
>>> from attrs import define, has
>>> from cattrs.gen import make_dict_structure_fn

>>> c = cattrs.Converter()
>>> c.register_structure_hook_factory(
...     has,
...     lambda cl: make_dict_structure_fn(
...         cl, c, _cattrs_forbid_extra_keys=True, _cattrs_detailed_validation=False
...     )
... )

>>> @define
... class E:
...     an_int: int

>>> c.structure({"an_int": 1, "else": 2}, E)
Traceback (most recent call last):
...
cattrs.errors.ForbiddenExtraKeysError: Extra fields in constructor for E: else
```

A complex use case for hook factories is described over at *Using factory hooks*.

CHAPTER

SEVEN

WHAT YOU CAN UNSTRUCTURE AND HOW

Unstructuring is intended to convert high-level, structured Python data (like instances of complex classes) into simple, unstructured data (like dictionaries).

Unstructuring is simpler than structuring in that no target types are required. Simply provide an argument to `Converter.unstructure()` and *cattrs* will produce a result based on the registered unstructuring hooks. A number of default unstructuring hooks are documented here.

7.1 Primitive Types and Collections

Primitive types (integers, floats, strings...) are simply passed through. Collections are copied. There's relatively little value in unstructuring these types directly as they are already unstructured and third-party libraries tend to support them directly.

A useful use case for unstructuring collections is to create a deep copy of a complex or recursive collection.

```
>>> # A dictionary of strings to lists of tuples of floats.
>>> data = {'a': [[1.0, 2.0], [3.0, 4.0]]}

>>> copy = cattrs.unstructure(data)
>>> data == copy
True
>>> data is copy
False
```

7.1.1 Typed Dicts

TypedDicts unstructure into dictionaries, potentially unchanged (depending on the exact field types and registered hooks).

```
>>> from typing import TypedDict
>>> from datetime import datetime, timezone
>>> from cattrs import Converter

>>> class MyTypedDict(TypedDict):
...     a: datetime

>>> c = Converter()
>>> c.register_unstructure_hook(datetime, lambda d: d.timestamp())
```

(continues on next page)

(continued from previous page)

```
>>> c.unstructure({"a": datetime(1970, 1, 1, tzinfo=timezone.utc)}, unstructure_
↪as=MyTypedDict)
{'a': 0.0}
```

Generic TypedDicts work on Python 3.11 and later, since that is the first Python version that supports them in general.

On Python 3.7, using `typing_extensions.TypedDict` is required since `typing.TypedDict` doesn't exist there. On Python 3.8, using `typing_extensions.TypedDict` is recommended since `typing.TypedDict` doesn't support all necessary features, so certain combinations of subclassing, totality and `typing.Required` won't work.

Similar to attrs *classes*, unstructuring can be customized using `cattrs.gen.typeddicts.make_dict_unstructure_fn()`.

```
>>> from typing import TypedDict
>>> from cattrs import Converter
>>> from cattrs.gen import override
>>> from cattrs.gen.typeddicts import make_dict_unstructure_fn

>>> class MyTypedDict(TypedDict):
...     a: int
...     b: int

>>> c = Converter()
>>> c.register_unstructure_hook(
...     MyTypedDict,
...     make_dict_unstructure_fn(
...         MyTypedDict,
...         c,
...         a=override(omit=True)
...     )
... )

>>> c.unstructure({"a": 1, "b": 2}, unstructure_as=MyTypedDict)
{'b': 2}
```

See also:

Structuring TypedDicts.

New in version 23.1.0.

7.2 pathlib.Path

`pathlib.Path` objects are unstructured into their string value.

```
>>> from pathlib import Path

>>> cattrs.unstructure(Path("/root"))
'/root'
```

New in version 23.1.0.

7.3 Customizing Collection Unstructuring

Important: This feature is supported for Python 3.9 and later.

Sometimes it's useful to be able to override collection unstructuring in a generic way. A common example is using a JSON library that doesn't support sets, but expects lists and tuples instead.

Using ordinary unstructuring hooks for this is unwieldy due to the semantics of singledispatch; in other words, you'd need to register hooks for all specific types of set you're using (`set[int]`, `set[float]`, `set[str]`...), which is not useful.

Function-based hooks can be used instead, but come with their own set of challenges - they're complicated to write efficiently.

The `Converter` supports easy customizations of collection unstructuring using its `unstruct_collection_overrides` parameter. For example, to unstructure all sets into lists, try the following:

```
>>> from collections.abc import Set
>>> converter = cattrs.Converter(unstruct_collection_overrides={Set: list})

>>> converter.unstructure({1, 2, 3})
[1, 2, 3]
```

Going even further, the Converter contains heuristics to support the following Python types, in order of decreasing generality:

- `Sequence`, `MutableSequence`, `list`, `deque`, `tuple`
- `Set`, `frozenset`, `MutableSet`, `set`
- `Mapping`, `MutableMapping`, `dict`, `defaultdict`, `OrderedDict`, `Counter`

For example, if you override the unstructure type for `Sequence`, but not for `MutableSequence`, `list` or `tuple`, the override will also affect those types. An easy way to remember the rule:

- all `MutableSequence` s are `Sequence` s, so the override will apply
- all `list` s are `MutableSequence` s, so the override will apply
- all `tuple` s are `Sequence` s, so the override will apply

If, however, you override only `MutableSequence`, fields annotated as `Sequence` will not be affected (since not all sequences are mutable sequences), and fields annotated as tuples will not be affected (since tuples are not mutable sequences in the first place).

Similar logic applies to the set and mapping hierarchies.

Make sure you're using the types from `collections.abc` on Python 3.9+, and from `typing` on older Python versions.

7.3.1 typing.Final

PEP 591 Final attribute types (`Final[int]`) are supported and unstructured appropriately.

New in version 23.1.0.

See also:

Structuring Final.

7.4 typing.Annotated

Fields marked as `typing.Annotated[type, ...]` are supported and are matched using the first type present in the annotated type.

7.5 typing.NewType

NewTypes are supported and are unstructured according to the rules for their underlying type. Their hooks can also be overriden using `Converter.register_unstructure_hook()`.

New in version 22.2.0.

See also:

Structuring NewTypes.

Note: NewTypes are not supported by the legacy `BaseConverter`.

7.6 *attrs* Classes and Dataclasses

attrs classes and dataclasses are supported out of the box. `cattrs.Converters` support two unstructuring strategies:

- `UnstructureStrategy.AS_DICT` - similar to `attrs.asdict()`, unstructures *attrs* and dataclass instances into dictionaries. This is the default.
- `UnstructureStrategy.AS_TUPLE` - similar to `attrs.astuple()`, unstructures *attrs* and dataclass instances into tuples.

```
>>> @define
... class C:
...     a = field()
...     b = field()

>>> inst = C(1, 'a')

>>> converter = cattrs.Converter(unstruct_strat=cattrs.UnstructureStrategy.AS_TUPLE)

>>> converter.unstructure(inst)
(1, 'a')
```

7.7 Mixing and Matching Strategies

Converters publicly expose two helper methods, `Converter.unstructure_attrs_asdict()` and `Converter.unstructure_attrs_astuple()`. These methods can be used with custom unstructuring hooks to selectively apply one strategy to instances of particular classes.

Assume two nested *attrs* classes, `Inner` and `Outer`; instances of `Outer` contain instances of `Inner`. Instances of `Outer` should be unstructured as dictionaries, and instances of `Inner` as tuples. Here's how to do this.

```
>>> @define
... class Inner:
...     a: int

>>> @define
... class Outer:
...     i: Inner

>>> inst = Outer(i=Inner(a=1))

>>> converter = cattrs.Converter()
>>> converter.register_unstructure_hook(Inner, converter.unstructure_attrs_astuple)

>>> converter.unstructure(inst)
{'i': (1,)}
```

Of course, these methods can be used directly as well, without changing the converter strategy.

```
>>> @define
... class C:
...     a: int
...     b: str

>>> inst = C(1, 'a')

>>> converter = cattrs.Converter()

>>> converter.unstructure_attrs_astuple(inst)  # Default is AS_DICT.
(1, 'a')
```

7.8 Unstructuring Hook Factories

Hook factories operate one level higher than unstructuring hooks; unstructuring hooks are functions registered to a class or predicate, and hook factories are functions (registered via a predicate) that produce unstructuring hooks.

Unstructuring hooks factories are registered using `Converter.register_unstructure_hook_factory()`.

Here's a small example showing how to use factory hooks to skip unstructuring `init=False` attributes on all *attrs* classes.

```
>>> from attrs import define, has, field, fields
>>> from cattrs import override
>>> from cattrs.gen import make_dict_unstructure_fn
```

(continues on next page)

(continued from previous page)

```
>>> c = cattrs.Converter()
>>> c.register_unstructure_hook_factory(
...     has,
...     lambda cl: make_dict_unstructure_fn(
...         cl, c, **{a.name: override(omit=True) for a in fields(cl) if not a.init}
...     )
... )

>>> @define
... class E:
...     an_int: int
...     another_int: int = field(init=False)

>>> inst = E(1)
>>> inst.another_int = 5
>>> c.unstructure(inst)
{'an_int': 1}
```

A complex use case for hook factories is described over at *Using factory hooks*.

CHAPTER

EIGHT

STRATEGIES

cattrs ships with a number of *strategies* for customizing un/structuring behavior.

Strategies are prepackaged, high-level patterns for quickly and easily applying complex customizations to a converter.

8.1 Tagged Unions Strategy

Found at `cattrs.strategies.configure_tagged_union()`.

The *tagged union* strategy allows for un/structuring a union of classes by including an additional field (the *tag*) in the unstructured representation. Each tag value is associated with a member of the union.

```
>>> from cattrs.strategies import configure_tagged_union
>>> from cattrs import Converter
>>> converter = Converter()

>>> @define
... class A:
...     a: int

>>> @define
... class B:
...     b: str

>>> configure_tagged_union(A | B, converter)

>>> converter.unstructure(A(1), unstructure_as=A | B)
{'a': 1, '_type': 'A'}

>>> converter.structure({'a': 1, '_type': 'A'}, A | B)
A(a=1)
```

By default, the tag field name is `_type` and the tag value is the class name of the union member. Both the field name and value can be overriden.

The `tag_generator` parameter is a one-argument callable that will be called with every member of the union to generate a mapping of tag values to union members. Here are some common `tag_generator` uses:

Tag info available in	Recommended `tag_generator`
Name of the class	Use the default, or `lambda cl: cl.__name__`
A class variable (`classvar`)	`lambda cl: cl.classvar`
A dictionary (`mydict`)	`mydict.get` or `mydict.__getitem__`
An enum of possible values	Build a dictionary of classes to enum values and use it

The union members aren't required to be attrs classes or dataclasses, although those work automatically. They may be anything that cattrs can un/structure from/to a dictionary, for example a type with registered custom hooks.

A default member can be specified to be used if the tag is missing or is unknown. This is useful for evolving APIs in a backwards-compatible way; an endpoint taking class `A` can be changed to take `A | B` with `A` as the default (for old clients which do not send the tag).

This strategy only applies in the context of the union; the normal un/structuring hooks are left untouched. This also means union members can be reused in multiple unions easily.

```
# Unstructuring as a union.
>>> converter.unstructure(A(1), unstructure_as=A | B)
{'a': 1, '_type': 'A'}

# Unstructuring as just an `A`.
>>> converter.unstructure(A(1))
{'a': 1}
```

8.1.1 Real-life Case Study

The Apple App Store supports server callbacks, by which Apple sends a JSON payload to a URL of your choice. The payload can be interpreted as about a dozen different messages, based on the value of the `notificationType` field.

To keep the example simple we define two classes, one for the `REFUND` event and one for everything else.

```
@define
class Refund:
    originalTransactionId: str

@define
class OtherAppleNotification:
    notificationType: str

AppleNotification = Refund | OtherAppleNotification
```

Next, we use the *tagged unions* strategy to prepare our converter. The tag value for the `Refund` event is `REFUND`, and we can let the `OtherAppleNotification` class handle all the other cases. The `tag_generator` parameter is a callable, so we can give it the `get` method of a dictionary.

```
>>> c = Converter()
>>> configure_tagged_union(
...     AppleNotification,
...     c,
```

(continues on next page)

(continued from previous page)

```
...     tag_name="notificationType",
...     tag_generator={Refund: "REFUND"},
...     default=OtherAppleNotification
... )
```

The converter is now ready to start structuring Apple notifications.

```
>>> payload = {"notificationType": "REFUND", "originalTransactionId": "1"}
>>> notification = c.structure(payload, AppleNotification)

>>> match notification:
...     case Refund(txn_id):
...         print(f"Refund for {txn_id}!")
...     case OtherAppleNotification(not_type):
...         print("Can't handle this yet")
```

8.2 Include Subclasses Strategy

Found at `cattrs.strategies.include_subclasses()`.

The *include subclass* strategy allows the un/structuring of a base class to an instance of itself or one of its descendants. Conceptually with this strategy, each time an un/structure operation for the base class is asked, `cattrs` machinery replaces that operation as if the union of the base class and its descendants had been asked instead.

```
>>> from attrs import define
>>> from cattrs.strategies import include_subclasses
>>> from cattrs import Converter

>>> @define
... class Parent:
...     a: int

>>> @define
... class Child(Parent):
...     b: str

>>> converter = Converter()
>>> include_subclasses(Parent, converter)

>>> converter.unstructure(Child(a=1, b="foo"), unstructure_as=Parent)
{'a': 1, 'b': 'foo'}

>>> converter.structure({'a': 1, 'b': 'foo'}, Parent)
Child(a=1, b='foo')
```

In the example above, we asked to unstructure then structure a `Child` instance as the `Parent` class and in both cases we correctly obtained back the unstructured and structured versions of the `Child` instance. If we did not apply the `include_subclasses` strategy, this is what we would have obtained:

```
>>> converter_no_subclasses = Converter()

>>> converter_no_subclasses.unstructure(Child(a=1, b="foo"), unstructure_as=Parent)
{'a': 1}

>>> converter_no_subclasses.structure({'a': 1, 'b': 'foo'}, Parent)
Parent(a=1)
```

Without the application of the strategy, in both unstructure and structure operations, we received a `Parent` instance.

Note: The handling of subclasses is an opt-in feature for two main reasons:

- Performance. While small and probably negligeable in most cases the subclass handling incurs more function calls and has a performance impact.
- Customization. The specific handling of subclasses can be different from one situation to the other. In particular there is not apparent universal good defaults for disambiguating the union type. Consequently the decision is left to the user.

Warning: To work properly, all subclasses must be defined when the `include_subclasses` strategy is applied to a `converter`. If subclasses types are defined later, for instance in the context of a plug-in mechanism using inheritance, then those late defined subclasses will not be part of the subclasses union type and will not be un/structured as expected.

8.2.1 Customization

In the example shown in the previous section, the default options for `include_subclasses` work well because the `Child` class has an attribute that do not exist in the `Parent` class (the `b` attribute). The automatic union type disambiguation function which is based on finding unique fields for each type of the union works as intended.

Sometimes, more disambiguation customization is required. For instance, the unstructuring operation would have failed if `Child` did not have an extra attribute or if a sibling of `Child` had also a `b` attribute. For those cases, a callable of 2 positional arguments (a union type and a converter) defining a *tagged union strategy* can be passed to the `include_subclasses` strategy. `configure_tagged_union()` can be used as-is, but if you want to change its defaults, the partial function from the `functools` module in the standard library can come in handy.

```
>>> from functools import partial
>>> from attrs import define
>>> from cattrs.strategies import include_subclasses, configure_tagged_union
>>> from cattrs import Converter

>>> @define
... class Parent:
...     a: int

>>> @define
... class Child1(Parent):
...     b: str
```

(continues on next page)

(continued from previous page)

```
>>> @define
... class Child2(Parent):
...     b: int

>>> converter = Converter()
>>> union_strategy = partial(configure_tagged_union, tag_name="type_name")
>>> include_subclasses(Parent, converter, union_strategy=union_strategy)

>>> converter.unstructure(Child1(a=1, b="foo"), unstructure_as=Parent)
{'a': 1, 'b': 'foo', 'type_name': 'Child1'}

>>> converter.structure({'a': 1, 'b': 1, 'type_name': 'Child2'}, Parent)
Child2(a=1, b=1)
```

Other customizations available see are (see `include_subclasses()`):

- The exact list of subclasses that should participate to the union with the `subclasses` argument.
- Attribute overrides that permit the customization of attributes un/structuring like renaming an attribute.

Here is an example involving both customizations:

```
>>> from attrs import define
>>> from cattrs.strategies import include_subclasses
>>> from cattrs import Converter, override

>>> @define
... class Parent:
...     a: int

>>> @define
... class Child(Parent):
...     b: str

>>> converter = Converter()
>>> include_subclasses(
...     Parent,
...     converter,
...     subclasses=(Parent, Child),
...     overrides={"b": override(rename="c")}
... )

>>> converter.unstructure(Child(a=1, b="foo"), unstructure_as=Parent)
{'a': 1, 'c': 'foo'}

>>> converter.structure({'a': 1, 'c': 'foo'}, Parent)
Child(a=1, b='foo')
```

CHAPTER

NINE

VALIDATION

cattrs has a detailed validation mode since version 22.1.0, and this mode is enabled by default. When running under detailed validation, the un/structuring hooks are slightly slower but produce more precise and exhaustive error messages.

9.1 Detailed Validation

New in version 22.1.0.

In detailed validation mode, any un/structuring errors will be grouped and raised together as a `cattrs.BaseValidationError`, which is a PEP 654 ExceptionGroup. ExceptionGroups are special exceptions which contain lists of other exceptions, which may themselves be other ExceptionGroups. In essence, ExceptionGroups are trees of exceptions.

When un/structuring a class, *cattrs* will gather any exceptions on a field-by-field basis and raise them as a `cattrs.ClassValidationError`, which is a subclass of `BaseValidationError`.

When structuring sequences and mappings, *cattrs* will gather any exceptions on a key- or index-basis and raise them as a `cattrs.IterableValidationError`, which is a subclass of `BaseValidationError`.

The exceptions will also have their `__notes__` attributes set, as per PEP 678, showing the field, key or index for each inner exception.

A simple example involving a class containing a list and a dictionary:

```
@define
class Class:
    a_list: list[int]
    a_dict: dict[str, int]

>>> structure({"a_list": ["a"], "a_dict": {"str": "a"}}, Class)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 276, in␣
↪structure
  |     return self._structure_func.dispatch(cl)(obj, cl)
  |   File "<cattrs generated structure __main__.Class>", line 14, in structure_Class
  |     if errors: raise __c_cve('While structuring Class', errors, __cl)
  | cattrs.errors.ClassValidationError: While structuring Class
  +-+---------------- 1 ----------------
    | Exception Group Traceback (most recent call last):
    |   File "<cattrs generated structure __main__.Class>", line 5, in structure_Class
    |     res['a_list'] = __c_structure_a_list(o['a_list'], __c_type_a_list)
```

(continues on next page)

(continued from previous page)

```
    |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 457, in _
↪structure_list
    |     raise IterableValidationError(
    | cattrs.errors.IterableValidationError: While structuring list[int]
    | Structuring class Class @ attribute a_list
    +-+---------------- 1 ----------------
      | Traceback (most recent call last):
      |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 450, in _
↪structure_list
      |     res.append(handler(e, elem_type))
      |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 375, in _
↪structure_call
      |     return cl(obj)
      | ValueError: invalid literal for int() with base 10: 'a'
      | Structuring list[int] @ index 0
      +------------------------------------
    +---------------- 2 ----------------
    | Exception Group Traceback (most recent call last):
    |   File "<cattrs generated structure __main__.Class>", line 10, in structure_Class
    |     res['a_dict'] = __c_structure_a_dict(o['a_dict'], __c_type_a_dict)
    |   File "", line 17, in structure_mapping
    | cattrs.errors.IterableValidationError: While structuring dict
    | Structuring class Class @ attribute a_dict
    +-+---------------- 1 ----------------
      | Traceback (most recent call last):
      |   File "", line 5, in structure_mapping
      | ValueError: invalid literal for int() with base 10: 'a'
      | Structuring mapping value @ key 'str'
      +------------------------------------
```

9.1.1 Transforming Exceptions into Error Messages

New in version 23.1.0.

ExceptionGroup stack traces are great while you're developing, but sometimes a more compact representation of validation errors is better. *cattrs* provides a helper function, `cattrs.transform_error()`, which transforms validation errors into lists of error messages.

The example from the previous paragraph produces the following error messages:

```
>>> from cattrs import transform_error

>>> try:
...     structure({"a_list": ["a"], "a_dict": {"str": "a"}}, Class)
... except Exception as exc:
...     print(transform_error(exc))

[
    'invalid value for type, expected int @ $.a_list[0]',
    "invalid value for type, expected int @ $.a_dict['str']"
]
```

A small number of built-in exceptions are converted into error messages automatically. This can be further customized by providing `cattrs.transform_error()` with a function that it can use to turn individual, non-ExceptionGroup exceptions into error messages. A useful pattern is wrapping the default, `cattrs.v.format_exception()` function.

```
>>> from cattrs.v iomport format_exception

>>> def my_exception_formatter(exc: BaseException, type) -> str:
...     if isinstance(exc, MyInterestingException):
...         return "My error message"
...     return format_exception(exc, type)

>>> try:
...     structure(..., Class)
... except Exception as exc:
...     print(transform_error(exc, format_exception=my_exception_formatter))
```

If even more customization is required, `cattrs.transform_error()` can be copied over into your codebase and adjusted as needed.

9.2 Non-detailed Validation

Non-detailed validation can be enabled by initializing any of the converters with `detailed_validation=False`. In this mode, any errors during un/structuring will bubble up directly as soon as they happen.

CHAPTER

TEN

PRECONFIGURED CONVERTERS

The `cattrs.preconf` package contains factories for preconfigured converters, specifically adjusted for particular serialization libraries.

For example, to get a converter configured for BSON:

```
>>> from cattrs.preconf.bson import make_converter

>>> converter = make_converter() # Takes the same parameters as the `cattrs.Converter`
```

Converters obtained this way can be customized further, just like any other converter.

These converters support the following classes and type annotations, both for structuring and unstructuring:

- `str`, `bytes`, `int`, `float`, `pathlib.Path` int enums, string enums
- *attrs* classes and dataclasses
- lists, homogenous tuples, heterogenous tuples, dictionaries, counters, sets, frozensets
- optionals
- sequences, mutable sequences, mappings, mutable mappings, sets, mutable sets
- `datetime.datetime`

New in version 22.1.0: All preconf converters now have `loads` and `dumps` methods, which combine un/structuring and the de/serialization logic from their underlying libraries.

```
>>> from cattrs.preconf.json import make_converter

>>> converter = make_converter()

>>> @define
... class Test:
...     a: int

>>> converter.dumps(Test(1))
'{"a": 1}'
```

Particular libraries may have additional constraints documented below.

Third-party libraries can be specified as optional (extra) dependencies on `cattrs` during installation. Optional install targets should match the name of the `cattrs.preconf` modules.

```
# Using pip
pip install cattrs[ujson]

# Using poetry
poetry add --extras tomlkit cattrs
```

10.1 Standard Library *json*

Found at `cattrs.preconf.json`.

Bytes are serialized as base 85 strings. Counters are serialized as dictionaries. Sets are serialized as lists, and deserialized back into sets. `datetime` s are serialized as ISO 8601 strings.

10.2 *ujson*

Found at `cattrs.preconf.ujson`.

Bytes are serialized as base 85 strings. Sets are serialized as lists, and deserialized back into sets. `datetime` s are serialized as ISO 8601 strings.

`ujson` doesn't support integers less than -9223372036854775808, and greater than 9223372036854775807, nor does it support `float('inf')`.

10.3 *orjson*

Found at `cattrs.preconf.orjson`.

Bytes are serialized as base 85 strings. Sets are serialized as lists, and deserialized back into sets. `datetime` s are serialized as ISO 8601 strings.

orjson doesn't support integers less than -9223372036854775808, and greater than 9223372036854775807. *orjson* only supports mappings with string keys so mappings will have their keys stringified before serialization, and de-stringified during deserialization.

10.4 *msgpack*

Found at `cattrs.preconf.msgpack`.

Sets are serialized as lists, and deserialized back into sets. `datetime` s are serialized as UNIX timestamp float values.

msgpack doesn't support integers less than -9223372036854775808, and greater than 18446744073709551615.

When parsing msgpack data from bytes, the library needs to be passed `strict_map_key=False` to get the full range of compatibility.

10.5 *cbor2*

New in version 23.1.0.

Found at `cattrs.preconf.cbor2`.

cbor2 implements a fully featured CBOR encoder with several extensions for handling shared references, big integers, rational numbers and so on.

Sets are serialized and deserialized to sets. Tuples are serialized as lists.

`datetime` s are serialized as a text string by default (CBOR Tag 0). Use keyword argument `datetime_as_timestamp=True` to encode as UNIX timestamp integer/float (CBOR Tag 1) **note:** this replaces timezone information as UTC.

Use keyword argument `canonical=True` for efficient encoding to the smallest binary output.

Floats can be forced to smaller output by casting to lower-precision formats by casting to `numpy` floats (and back to Python floats). Example: `float(np.float32(value))` or `float(np.float16(value))`

10.6 *bson*

Found at `cattrs.preconf.bson`. Tested against the *bson* module bundled with the *pymongo* library, not the standalone PyPI *bson* package.

Sets are serialized as lists, and deserialized back into sets.

bson doesn't support integers less than -9223372036854775808 or greater than 9223372036854775807 (64-bit signed). *bson* does not support null bytes in mapping keys. *bson* only supports mappings with string keys so mappings will have their keys stringified before serialization, and destringified during deserialization. The *bson* datetime representation doesn't support microsecond accuracy.

When encoding and decoding, the library needs to be passed `codec_options=bson.CodecOptions(tz_aware=True)` to get the full range of compatibility.

10.7 *pyyaml*

Found at `cattrs.preconf.pyyaml`.

Frozensets are serialized as lists, and deserialized back into frozensets.

10.8 *tomlkit*

Found at `cattrs.preconf.tomlkit`.

Bytes are serialized as base 85 strings. Sets are serialized as lists, and deserialized back into sets. Tuples are serialized as lists, and deserialized back into tuples. *tomlkit* only supports mappings with string keys so mappings will have their keys stringified before serialization, and destringified during deserialization.

CHAPTER

ELEVEN

CUSTOMIZING CLASS UN/STRUCTURING

This section deals with customizing the unstructuring and structuring processes in `cattrs`.

11.1 Using `cattr.Converter`

The default `Converter`, upon first encountering an `attrs` class, will use the generation functions mentioned here to generate the specialized hooks for it, register the hooks and use them.

11.2 Manual un/structuring hooks

You can write your own structuring and unstructuring functions and register them for types using `Converter.register_structure_hook` and `Converter.register_unstructure_hook`. This approach is the most flexible but also requires the most amount of boilerplate.

11.3 Using `cattrs.gen` generators

`cattrs` includes a module, `cattrs.gen`, which allows for generating and compiling specialized functions for unstructuring `attrs` classes.

One reason for generating these functions in advance is that they can bypass a lot of `cattrs` machinery and be significantly faster than normal `cattrs`.

Another reason is that it's possible to override behavior on a per-attribute basis.

Currently, the overrides only support generating dictionary un/structuring functions (as opposed to tuples), and support `omit_if_default`, `forbid_extra_keys`, `rename` and `omit`.

11.3.1 `omit_if_default`

This override can be applied on a per-class or per-attribute basis. The generated unstructuring function will skip unstructuring values that are equal to their default or factory values.

```
>>> from cattrs.gen import make_dict_unstructure_fn, override
>>>
>>> @define
... class WithDefault:
...     a: int
```

(continues on next page)

(continued from previous page)

```
...     b: dict = Factory(dict)
>>>
>>> c = cattrs.Converter()
>>> c.register_unstructure_hook(WithDefault, make_dict_unstructure_fn(WithDefault, c,␣
↪b=override(omit_if_default=True)))
>>> c.unstructure(WithDefault(1))
{'a': 1}
```

Note that the per-attribute value overrides the per-class value. A side-effect of this is the ability to force the presence of a subset of fields. For example, consider a class with a `DateTime` field and a factory for it: skipping the unstructuring of the `DateTime` field would be inconsistent and based on the current time. So we apply the `omit_if_default` rule to the class, but not to the `DateTime` field.

Note:

```
The parameter to `make_dict_unstructure_function` is named ``_cattrs_omit_if_default``␣
↪instead of just ``omit_if_default`` to avoid potential collisions with an override for␣
↪a field named ``omit_if_default``.
```

```
>>> from pendulum import DateTime
>>> from cattrs.gen import make_dict_unstructure_fn, override
>>>
>>> @define
... class TestClass:
...     a: Optional[int] = None
...     b: DateTime = Factory(DateTime.utcnow)
>>>
>>> c = cattrs.Converter()
>>> hook = make_dict_unstructure_fn(TestClass, c, _cattrs_omit_if_default=True,␣
↪b=override(omit_if_default=False))
>>> c.register_unstructure_hook(TestClass, hook)
>>> c.unstructure(TestClass())
{'b': ...}
```

This override has no effect when generating structuring functions.

11.3.2 forbid_extra_keys

By default `cattrs` is lenient in accepting unstructured input. If extra keys are present in a dictionary, they will be ignored when generating a structured object. Sometimes it may be desirable to enforce a stricter contract, and to raise an error when unknown keys are present - in particular when fields have default values this may help with catching typos. `forbid_extra_keys` can also be enabled (or disabled) on a per-class basis when creating structure hooks with `make_dict_structure_fn`.

```
>>> from cattrs.gen import make_dict_structure_fn
>>>
>>> @define
... class TestClass:
...     number: int = 1
>>>
```

(continues on next page)

(continued from previous page)

```
>>> c = cattrs.Converter(forbid_extra_keys=True)
>>> c.structure({"nummber": 2}, TestClass)
Traceback (most recent call last):
...
ForbiddenExtraKeyError: Extra fields in constructor for TestClass: nummber
>>> hook = make_dict_structure_fn(TestClass, c, _cattrs_forbid_extra_keys=False)
>>> c.register_structure_hook(TestClass, hook)
>>> c.structure({"nummber": 2}, TestClass)
TestClass(number=1)
```

This behavior can only be applied to classes or to the default for the `Converter`, and has no effect when generating unstructuring functions.

11.3.3 rename

Using the rename override makes `cattrs` simply use the provided name instead of the real attribute name. This is useful if an attribute name is a reserved keyword in Python.

```
>>> from pendulum import DateTime
>>> from cattrs.gen import make_dict_unstructure_fn, make_dict_structure_fn, override
>>>
>>> @define
... class ExampleClass:
...     klass: Optional[int]
>>>
>>> c = cattrs.Converter()
>>> unst_hook = make_dict_unstructure_fn(ExampleClass, c, klass=override(rename="class"))
>>> st_hook = make_dict_structure_fn(ExampleClass, c, klass=override(rename="class"))
>>> c.register_unstructure_hook(ExampleClass, unst_hook)
>>> c.register_structure_hook(ExampleClass, st_hook)
>>> c.unstructure(ExampleClass(1))
{'class': 1}
>>> c.structure({'class': 1}, ExampleClass)
ExampleClass(klass=1)
```

11.3.4 omit

This override can only be applied to individual attributes. Using the `omit` override will simply skip the attribute completely when generating a structuring or unstructuring function.

```
>>> from cattrs.gen import make_dict_unstructure_fn, override
>>>
>>> @define
... class ExampleClass:
...     an_int: int
>>>
>>> c = cattrs.Converter()
>>> unst_hook = make_dict_unstructure_fn(ExampleClass, c, an_int=override(omit=True))
>>> c.register_unstructure_hook(ExampleClass, unst_hook)
>>> c.unstructure(ExampleClass(1))
{}
```

11.3.5 `struct_hook` and `unstruct_hook`

By default, the generators will determine the right un/structure hook for each attribute of a class at time of generation according to the type of each individual attribute.

This process can be overriden by passing in the desired un/structure manually.

```
>>> from cattrs.gen import make_dict_structure_fn, override

>>> @define
... class ExampleClass:
...     an_int: int

>>> c = cattrs.Converter()
>>> st_hook = make_dict_structure_fn(
...     ExampleClass, c, an_int=override(struct_hook=lambda v, _: v + 1)
... )
>>> c.register_structure_hook(ExampleClass, st_hook)

>>> c.structure({"an_int": 1}, ExampleClass)
ExampleClass(an_int=2)
```

CHAPTER

TWELVE

TIPS FOR HANDLING UNIONS

This sections contains information for advanced union handling.

As mentioned in the structuring section, *cattrs* is able to handle simple unions of *attrs* classes automatically. More complex cases require converter customization (since there are many ways of handling unions).

12.1 Unstructuring unions with extra metadata

Note: *cattrs* comes with the *tagged unions strategy* for handling this exact use-case since version 23.1. The example below has been left here for educational purposes, but you should prefer the strategy.

Let's assume a simple scenario of two classes, `ClassA` and `ClassB`, both of which have no distinct fields and so cannot be used automatically with *cattrs*.

```
@define
class ClassA:
    a_string: str

@define
class ClassB:
    a_string: str
```

A naive approach to unstructuring either of these would yield identical dictionaries, and not enough information to restructure the classes.

```
>>> converter.unstructure(ClassA("test"))
{'a_string': 'test'}  # Is this ClassA or ClassB? Who knows!
```

What we can do is ensure some extra information is present in the unstructured data, and then use that information to help structure later.

First, we register an unstructure hook for the `Union[ClassA, ClassB]` type.

```
>>> converter.register_unstructure_hook(
...     Union[ClassA, ClassB],
...     lambda o: {"_type": type(o).__name__,  **converter.unstructure(o)}
... )
>>> converter.unstructure(ClassA("test"), unstructure_as=Union[ClassA, ClassB])
{'_type': 'ClassA', 'a_string': 'test'}
```

Note that when unstructuring, we had to provide the `unstructure_as` parameter or *cattrs* would have just applied the usual unstructuring rules to `ClassA`, instead of our special union hook.

Now that the unstructured data contains some information, we can create a structuring hook to put it to use:

```
>>> converter.register_structure_hook(
...     Union[ClassA, ClassB],
...     lambda o, _: converter.structure(o, ClassA if o["_type"] == "ClassA" else ClassB)
... )
>>> converter.structure({"_type": "ClassA", "a_string": "test"}, Union[ClassA, ClassB])
ClassA(a_string='test')
```

CHAPTER

THIRTEEN

BENCHMARKING

cattrs includes a benchmarking suite to help detect performance regressions and guide performance optimizations.

The suite is based on pytest and pytest-benchmark. Benchmarks are similar to tests, with the exception of being stored in the `bench/` directory and being used to verify performance instead of correctness.

13.1 A Sample Workflow

First, ensure the system you're benchmarking on is as stable as possible. For example, the pyperf library has a `system tune` command that can tweak CPU frequency governors. You also might want to quit as many applications as possible and run the benchmark suite on isolated CPU cores (`taskset` can be used for this purpose on Linux).

Then, generate a baseline using `make bench`. This will run the benchmark suite and save it into a file.

Following that, implement the changes you have in mind. Run the test suite to ensure correctness. Then, compare the performance of the new code to the saved baseline using `make bench-cmp`. If the code is still correct but faster, congratulations!

CHAPTER

FOURTEEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

14.1 Types of Contributions

14.1.1 Report Bugs

Report bugs at https://github.com/python-attrs/cattrs/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

14.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

14.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

14.1.4 Write Documentation

cattrs could always use more documentation, whether as part of the official cattrs docs, in docstrings, or even on the web in blog posts, articles, and such.

14.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/python-attrs/cattrs/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

14.2 Get Started!

Ready to contribute? Here's how to set up *cattrs* for local development.

1. Fork the `cattrs` repo on GitHub.
2. Clone your fork locally::

```
$ git clone git@github.com:your_name_here/cattrs.git
```

3. Install your local copy into a virtualenv. Assuming you have poetry installed, this is how you set up your fork for local development::

```
$ cd cattrs/
$ poetry install --all-extras
```

4. Create a branch for local development::

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox::

```
$ poetry shell
$ make lint
$ make test
$ tox
```

6. Commit your changes and push your branch to GitHub::

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

14.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for all supported Python versions. Check https://github.com/python-attrs/cattrs/actions and make sure that the tests pass for all supported Python versions.

14.4 Tips

To run a subset of tests:

```
$ pytest tests.test_unstructure
```

CHAPTER

FIFTEEN

HISTORY

15.1 23.1.2 (2023-06-02)

- Improve `typing_extensions` version bound. (#372)

15.2 23.1.1 (2023-05-30)

- Add `typing_extensions` as a direct dependency on 3.10. (#369 #370)

15.3 23.1.0 (2023-05-30)

- Introduce the `tagged_union` strategy. (#318 #317)
- Introduce the `cattrs.transform_error` helper function for formatting validation exceptions. (258 342)
- Add support for `typing.TypedDict` and `typing_extensions.TypedDict`. (#296 #364)
- Add support for `typing.Final`. (#340 #349)
- Introduce `override.struct_hook` and `override.unstruct_hook`. Learn more here. (#326)
- Fix generating structuring functions for types with angle brackets (<>) and pipe symbols (|) in the name. (#319 #327)
- `pathlib.Path` is now supported by default. (#81)
- Add `cbor2` serialization library to the `cattr.preconf` package.
- Add optional dependencies for `cattrs.preconf` third-party libraries. (#337)
- All preconf converters now allow overriding the default `unstruct_collection_overrides` in `make_converter`. (#350 #353)
- Subclasses structuring and unstructuring is now supported via a custom `include_subclasses` strategy. (#312)
- Add support for `typing_extensions.Annotated` when the python version is less than `3.9`. (#366)
- Add unstructuring and structuring support for the standard library `deque`. (#355)

15.4 22.2.0 (2022-10-03)

- *Potentially breaking*: `cattrs.Converter` has been renamed to `cattrs.BaseConverter`, and `cattrs.GenConverter` to `cattrs.Converter`. The `GenConverter` name is still available for backwards compatibility, but is deprecated. If you were depending on functionality specific to the old `Converter`, change your import to `from cattrs import BaseConverter`.
- NewTypes are now supported by the `cattrs.Converter`. (#255 #94 #297)
- `cattrs.Converter` and `cattrs.BaseConverter` can now copy themselves using the `copy` method. (#284)
- Python 3.11 support.
- cattrs now supports un/structuring `kw_only` fields on attrs classes into/from dictionaries. (#247)
- PyPy support (and tests, using a minimal Hypothesis profile) restored. (#253)
- Fix propagating the `detailed_validation` flag to mapping and counter structuring generators.
- Fix `typing.Set` applying too broadly when used with the `GenConverter.unstruct_collection_overrides` parameter on Python versions below 3.9. Switch to `typing.AbstractSet` on those versions to restore the old behavior. (#264)
- Uncap the required Python version, to avoid problems detailed here (#275)
- Fix `Converter.register_structure_hook_factory` and `cattrs.gen.make_dict_unstructure_fn` type annotations. (#281)
- Expose all error classes in the `cattr.errors` namespace. Note that it is deprecated, just use `cattrs.errors`. (#252)
- Fix generating structuring functions for types with quotes in the name. (#291 #277)
- Fix usage of notes for the final version of PEP 678, supported since `exceptiongroup>=1.0.0rc4`. (#303)

15.5 22.1.0 (2022-04-03)

- *cattrs* now uses the CalVer versioning convention.
- *cattrs* now has a detailed validation mode, which is enabled by default. Learn more here. The old behavior can be restored by creating the converter with `detailed_validation=False`.
- *attrs* and dataclass structuring is now ~25% faster.
- Fix an issue structuring bare `typing.List` s on Pythons lower than 3.9. (#209)
- Fix structuring of non-parametrized containers like `list/dict/...` on Pythons lower than 3.9. (#218)
- Fix structuring bare `typing.Tuple` on Pythons lower than 3.9. (#218)
- Fix a wrong `AttributeError` of an missing `__parameters__` attribute. This could happen when inheriting certain generic classes – for example `typing.*` classes are affected. (#217)
- Fix structuring of `enum.Enum` instances in `typing.Literal` types. (#231)
- Fix unstructuring all tuples - unannotated, variable-length, homogenous and heterogenous - to `list`. (#226)
- For `forbid_extra_keys` raise custom `ForbiddenExtraKeyError` instead of generic `Exception`. (#225)
- All preconf converters now support `loads` and `dumps` directly. See an example here.
- Fix mappings with byte keys for the orjson, bson and tomlkit converters. (#241)

15.6 1.10.0 (2022-01-04)

- Add PEP 563 (string annotations) support for dataclasses. (#195)
- Fix handling of dictionaries with string Enum keys for bson, orjson, and tomlkit.
- Rename the `cattr.gen.make_dict_unstructure_fn.omit_if_default` parameter to `_cattrs_omit_if_default`, for consistency. The `omit_if_default` parameters to `GenConverter` and `override` are unchanged.
- Following the changes in *attrs* 21.3.0, add a `cattrs` package mirroring the existing `cattr` package. Both package names may be used as desired, and the `cattr` package isn't going away.

15.7 1.9.0 (2021-12-06)

- Python 3.10 support, including support for the new union syntax (`A | B` vs `Union[A, B]`).
- The `GenConverter` can now properly structure generic classes with generic collection fields. (#149)
- `omit=True` now also affects generated structuring functions. (#166)
- `cattr.gen.{make_dict_structure_fn, make_dict_unstructure_fn}` now resolve type annotations automatically when PEP 563 is used. (#169)
- Protocols are now unstructured as their runtime types. (#177)
- Fix an issue generating structuring functions with renaming and `_cattrs_forbid_extra_keys=True`. (#190)

15.8 1.8.0 (2021-08-13)

- Fix `GenConverter` mapping structuring for unannotated dicts on Python 3.8. (#151)
- The source code for generated un/structuring functions is stored in the `linecache` cache, which enables more informative stack traces when un/structuring errors happen using the `GenConverter`. This behavior can optionally be disabled to save memory.
- Support using the attr converter callback during structure. By default, this is a method of last resort, but it can be elevated to the default by setting `prefer_attrib_converters=True` on `Converter` or `GenConverter`. (#138)
- Fix structuring recursive classes. (#159)
- Converters now support un/structuring hook factories. This is the most powerful and complex venue for customizing un/structuring. This had previously been an internal feature.
- The Common Usage Examples documentation page now has a section on advanced hook factory usage.
- `cattr.override` now supports the `omit` parameter, which makes *cattrs* skip the atribute entirely when unstructuring.
- The `cattr.preconf.bson` module is now tested against the `bson` module bundled with the `pymongo` package, because that package is much more popular than the standalone PyPI `bson` package.

15.9 1.7.1 (2021-05-28)

- `Literal` s are not supported on Python 3.9.0 (supported on 3.9.1 and later), so we skip importing them there. (#150)

15.10 1.7.0 (2021-05-26)

- `cattr.global_converter` (which provides `cattr.unstructure`, `cattr.structure` etc.) is now an instance of `cattr.GenConverter`.
- `Literal` s are now supported and validated when structuring.
- Fix dependency metadata information for *attrs*. (#147)
- Fix `GenConverter` mapping structuring for unannotated dicts. (#148)

15.11 1.6.0 (2021-04-28)

- *cattrs* now uses Poetry.
- `GenConverter` mapping structuring is now ~25% faster, and unstructuring heterogenous tuples is significantly faster.
- Add `cattr.preconf`. This package contains modules for making converters for particular serialization libraries. We currently support the standard library `json`, and third-party `ujson`, `orjson`, `msgpack`, `bson`, `pyyaml` and `tomlkit` libraries.

15.12 1.5.0 (2021-04-15)

- Fix an issue with `GenConverter` unstructuring *attrs* classes and dataclasses with generic fields. (#65)
- `GenConverter` has support for easy overriding of collection unstructuring types (for example, unstructure all sets to lists) through its `unstruct_collection_overrides` argument. (#137)
- Unstructuring mappings with `GenConverter` is significantly faster.
- `GenConverter` supports strict handling of unexpected dictionary keys through its `forbid_extra_keys` argument. (#142)

15.13 1.4.0 (2021-03-21)

- Fix an issue with `GenConverter` un/structuring hooks when a function hook is registered after the converter has already been used.
- Add support for `collections.abc.{Sequence, MutableSequence, Set, MutableSet}`. These should be used on 3.9+ instead of their `typing` alternatives, which are deprecated. (#128)
- The `GenConverter` will unstructure iterables (`list[T]`, `tuple[T, ...]`, `set[T]`) using their type argument instead of the runtime class if its elements, if possible. These unstructuring operations are up to 40% faster. (#129)
- Flesh out `Converter` and `GenConverter` initializer type annotations. (#131)

- Add support for `typing.Annotated` on Python 3.9+. *cattrs* will use the first annotation present. *cattrs* specific annotations may be added in the future. (#127)
- Add support for dataclasses. (#43)

15.14 1.3.0 (2021-02-25)

- *cattrs* now has a benchmark suite to help make and keep cattrs the fastest it can be. The instructions on using it can be found under the Benchmarking section in the docs. (#123)
- Fix an issue unstructuring tuples of non-primitives. (#125)
- *cattrs* now calls `attr.resolve_types` on *attrs* classes when registering un/structuring hooks.
- `GenConverter` structuring and unstructuring of *attrs* classes is significantly faster.

15.15 1.2.0 (2021-01-31)

- `converter.unstructure` now supports an optional parameter, `unstructure_as`, which can be used to unstructure something as a different type. Useful for unions.
- Improve support for union un/structuring hooks. Flesh out docs for advanced union handling. (#115)
- Fix `GenConverter` behavior with inheritance hierarchies of *attrs* classes. ([#117](https://github.com/python-attrs/cattrs/pull/117 #116)
- Refactor `GenConverter.un/structure_attrs_fromdict` into `GenConverter.gen_un/structure_attrs_fromdict` to allow calling back to `Converter.un/structure_attrs_fromdict` without sideeffects. (#118)

15.16 1.1.2 (2020-11-29)

- The default disambiguator will not consider non-required fields any more. (#108)
- Fix a couple type annotations. (#107 #105)
- Fix a `GenConverter` unstructuring issue and tests.

15.17 1.1.1 (2020-10-30)

- Add metadata for supported Python versions. (#103)

15.18 1.1.0 (2020-10-29)

- Python 2, 3.5 and 3.6 support removal. If you need it, use a version below 1.1.0.
- Python 3.9 support, including support for built-in generic types (`list[int]` vs `typing.List[int]`).
- *cattrs* now includes functions to generate specialized structuring and unstructuring hooks. Specialized hooks are faster and support overrides (`omit_if_default` and `rename`). See the `cattr.gen` module.
- *cattrs* now includes a converter variant, `cattr.GenConverter`, that automatically generates specialized hooks for attrs classes. This converter will become the default in the future.
- Generating specialized structuring hooks now invokes attr.resolve_types on a class if the class makes use of the new PEP 563 annotations.
- *cattrs* now depends on *attrs* >= 20.1.0, because of `attr.resolve_types`.
- Specialized hooks now support generic classes. The default converter will generate and use a specialized hook upon encountering a generic class.

15.19 1.0.0 (2019-12-27)

- *attrs* classes with private attributes can now be structured by default.
- Structuring from dictionaries is now more lenient: extra keys are ignored.
- *cattrs* has improved type annotations for use with Mypy.
- Unstructuring sets and frozensets now works properly.

15.20 0.9.1 (2019-10-26)

- Python 3.8 support.

15.21 0.9.0 (2018-07-22)

- Python 3.7 support.

15.22 0.8.1 (2018-06-19)

- The disambiguation function generator now supports unions of *attrs* classes and NoneType.

15.23 0.8.0 (2018-04-14)

- Distribution fix.

15.24 0.7.0 (2018-04-12)

- Removed the undocumented `Converter.unstruct_strat` property setter.
- Removed the ability to set the `Converter.structure_attrs` instance field.
- Some micro-optimizations were applied; a `structure(unstructure(obj))` roundtrip is now up to 2 times faster.

15.25 0.6.0 (2017-12-25)

- Packaging fixes. (#17)

15.26 0.5.0 (2017-12-11)

- structure/unstructure now supports using functions as well as classes for deciding the appropriate function.
- added `Converter.register_structure_hook_func`, to register a function instead of a class for determining handler func.
- added `Converter.register_unstructure_hook_func`, to register a function instead of a class for determining handler func.
- vendored typing is no longer needed, nor provided.
- Attributes with default values can now be structured if they are missing in the input. (#15)
- `Optional` attributes can no longer be structured if they are missing in the input.
- `cattr.typed` removed since the functionality is now present in *attrs* itself. Replace instances of `cattr.typed(type)` with `attr.ib(type=type)`.

15.27 0.4.0 (2017-07-17)

- `Converter.loads` is now `Converter.structure`, and `Converter.dumps` is now `Converter.unstructure`.
- Python 2.7 is supported.
- Moved `cattr.typing` to `cattr.vendor.typing` to support different vendored versions of typing.py for Python 2 and Python 3.
- Type metadata can be added to *attrs* classes using `cattr.typed`.

15.28 0.3.0 (2017-03-18)

- Python 3.4 is no longer supported.
- Introduced `cattr.typing` for use with Python versions 3.5.2 and 3.6.0.
- Minor changes to work with newer versions of `typing`.
- Bare Optionals are not supported any more (use `Optional[Any]`).
- Attempting to load unrecognized classes will result in a ValueError, and a helpful message to register a loads hook.
- Loading *attrs* classes is now documented.
- The global converter is now documented.
- `cattr.loads_attrs_fromtuple` and `cattr.loads_attrs_fromdict` are now exposed.

15.29 0.2.0 (2016-10-02)

- Tests and documentation.

15.30 0.1.0 (2016-08-13)

- First release on PyPI.

CHAPTER

SIXTEEN

CATTRS

cattrs is an open source Python library for structuring and unstructuring data. *cattrs* works best with *attrs* classes, dataclasses and the usual Python collections, but other kinds of classes are supported by manually registering converters.

Python has a rich set of powerful, easy to use, built-in data types like dictionaries, lists and tuples. These data types are also the lingua franca of most data serialization libraries, for formats like json, msgpack, cbor, yaml or toml.

Data types like this, and mappings like `dict` s in particular, represent unstructured data. Your data is, in all likelihood, structured: not all combinations of field names or values are valid inputs to your programs. In Python, structured data is better represented with classes and enumerations. *attrs* is an excellent library for declaratively describing the structure of your data, and validating it.

When you're handed unstructured data (by your network, file system, database...), *cattrs* helps to convert this data into structured data. When you have to convert your structured data into data types other libraries can handle, *cattrs* turns your classes and enumerations into dictionaries, integers and strings.

Here's a simple taste. The list containing a float, an int and a string gets converted into a tuple of three ints.

```
>>> import cattrs

>>> cattrs.structure([1.0, 2, "3"], tuple[int, int, int])
(1, 2, 3)
```

cattrs works well with *attrs* classes out of the box.

```
>>> from attrs import frozen
>>> import cattrs

>>> @frozen  # It works with non-frozen classes too.
... class C:
...     a: int
...     b: str

>>> instance = C(1, 'a')
>>> cattrs.unstructure(instance)
{'a': 1, 'b': 'a'}
>>> cattrs.structure({'a': 1, 'b': 'a'}, C)
C(a=1, b='a')
```

Here's a much more complex example, involving `attrs` classes with type metadata.

```
>>> from enum import unique, Enum
>>> from typing import Optional, Sequence, Union
>>> from cattrs import structure, unstructure
>>> from attrs import define, field

>>> @unique
... class CatBreed(Enum):
...     SIAMESE = "siamese"
...     MAINE_COON = "maine_coon"
...     SACRED_BIRMAN = "birman"

>>> @define
... class Cat:
...     breed: CatBreed
...     names: Sequence[str]

>>> @define
... class DogMicrochip:
...     chip_id = field()  # Type annotations are optional, but recommended
...     time_chipped: float = field()

>>> @define
... class Dog:
...     cuteness: int
...     chip: Optional[DogMicrochip] = None

>>> p = unstructure([Dog(cuteness=1, chip=DogMicrochip(chip_id=1, time_chipped=10.0)),
...                  Cat(breed=CatBreed.MAINE_COON, names=('Fluffly', 'Fluffer'))])

>>> print(p)
[{'cuteness': 1, 'chip': {'chip_id': 1, 'time_chipped': 10.0}}, {'breed': 'maine_coon',
↪'names': ('Fluffly', 'Fluffer')}]
>>> print(structure(p, list[Union[Dog, Cat]]))
[Dog(cuteness=1, chip=DogMicrochip(chip_id=1, time_chipped=10.0)), Cat(breed=<CatBreed.
↪MAINE_COON: 'maine_coon'>, names=['Fluffly', 'Fluffer'])]
```

Consider unstructured data a low-level representation that needs to be converted to structured data to be handled, and use `structure`. When you're done, `unstructure` the data to its unstructured form and pass it along to another library or module. Use attrs type metadata to add type metadata to attributes, so *cattrs* will know how to structure and destructure them.

- Free software: MIT license
- Documentation: https://catt.rs
- Python versions supported: 3.7 and up. (Older Python versions, like 2.7, 3.5 and 3.6 are supported by older versions; see the changelog.)

16.1 Features

- Converts structured data into unstructured data, recursively:
  - *attrs* classes and dataclasses are converted into dictionaries in a way similar to `attrs.asdict`, or into tuples in a way similar to `attrs.astuple`.
  - Enumeration instances are converted to their values.
  - Other types are let through without conversion. This includes types such as integers, dictionaries, lists and instances of non-*attrs* classes.
  - Custom converters for any type can be registered using `register_unstructure_hook`.
- Converts unstructured data into structured data, recursively, according to your specification given as a type. The following types are supported:
  - `typing.Optional[T]`.
  - `typing.List[T]`, `typing.MutableSequence[T]`, `typing.Sequence[T]` (converts to a list).
  - `typing.Tuple` (both variants, `Tuple[T, ...]` and `Tuple[X, Y, Z]`).
  - `typing.MutableSet[T]`, `typing.Set[T]` (converts to a set).
  - `typing.FrozenSet[T]` (converts to a frozenset).
  - `typing.Dict[K, V]`, `typing.MutableMapping[K, V]`, `typing.Mapping[K, V]` (converts to a dict).
  - *attrs* classes with simple attributes and the usual `__init__`.
    - Simple attributes are attributes that can be assigned unstructured data, like numbers, strings, and collections of unstructured data.
  - All *attrs* classes and dataclasses with the usual `__init__`, if their complex attributes have type metadata.
  - `typing.Union` s of supported *attrs* classes, given that all of the classes have a unique field.
  - `typing.Union` s of anything, given that you provide a disambiguation function for it.
  - Custom converters for any type can be registered using `register_structure_hook`.

cattrs comes with preconfigured converters for a number of serialization libraries, including json, msgpack, cbor2, bson, yaml and toml. For details, see the cattr.preconf package.

16.2 Additional documentation and talks

- On structured and unstructured data, or the case for cattrs
- Why I use attrs instead of pydantic
- cattrs I: un/structuring speed
- Python has a macro language - it's Python (PyCon IT 2022)

16.3 Credits

Major credits to Hynek Schlawack for creating attrs and its predecessor, characteristic.

cattrs is tested with Hypothesis, by David R. MacIver.

cattrs is benchmarked using perf and pytest-benchmark.

This package was created with Cookiecutter and the `audreyr/cookiecutter-pypackage` project template.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex