Typeguard

Release 4.0.0

Alex Grönholm

Jul 12, 2023

CONTENTS

This library provides run-time type checking for functions defined with PEP 484 argument (and return) type annotations, and any arbitrary objects. It can be used together with static type checkers as an additional layer of type safety, to catch type violations that could only be detected at run time.

Two principal ways to do type checking are provided:

1. The `check_type` function:
   - like `isinstance()`, but supports arbitrary type annotations (within limits)
   - can be used as a `cast()` replacement, but with actual checking of the value
2. Code instrumentation:
   - entire modules, or individual functions (via `@typechecked`) are recompiled, with type checking code injected into them
   - automatically checks function arguments, return values and assignments to annotated local variables
   - for generator functions (regular and async), checks yield and send values
   - requires the original source code of the instrumented module(s) to be accessible

Two options are provided for code instrumentation:

1. the `@typechecked` function:
   - can be applied to functions individually
2. the import hook (`typeguard.install_import_hook()`):
   - automatically instruments targeted modules on import
   - no manual code changes required in the target modules
   - requires the import hook to be installed before the targeted modules are imported
   - may clash with other import hooks

CHAPTER

ONE

QUICK LINKS

1.1 User guide

1.1.1 Checking types directly

The most straightfoward way to do type checking with Typeguard is with *check_type()*. It can be used as as a beefed-up version of `isinstance()` that also supports checking against annotations in the `typing` module:

```
from typeguard import check_type

# Raises TypeCheckError if there's a problem
check_type([1234], List[int])
```

It's also useful for safely casting the types of objects dynamically constructed from external sources:

```
import json
from typing import List, TypedDict

from typeguard import check_type

# Example contents of "people.json":
# [
#   {"name": "John Smith", "phone": "111-123123", "address": "123 Main Street"},
#   {"name": "Jane Smith", "phone": "111-456456", "address": "123 Main Street"}
# ]

class Person(TypedDict):
    name: str
    phone: str
    address: str

 with open("people.json") as f:
    people = check_type(json.load(f), List[Person])
```

With this code, static type checkers will recognize the type of `people` to be `List[Person]`.

1.1.2 Using the decorator

The *@typechecked* decorator is the simplest way to add type checking on a case-by-case basis. It can be used on functions directly, or on entire classes, in which case all the contained methods are instrumented:

```
from typeguard import typechecked

@typechecked
def some_function(a: int, b: float, c: str, *args: str) -> bool:
    ...
    return retval

@typechecked
class SomeClass:
    # All type annotated methods (including static and class methods and properties)
    # are type checked.
    # Does not apply to inner classes!
    def method(x: int) -> int:
        ...
```

The decorator instruments functions by fetching the source code, parsing it to an abstract syntax tree using `ast.parse()`, modifying it to add type checking, and finally compiling the modified AST into byte code. This code is then used to make a new function object that is used to replace the original one.

To explicitly set type checking options on a per-function basis, you can pass them as keyword arguments to *@typechecked*:

```
from typeguard import CollectionCheckStrategy, typechecked

@typechecked(collection_check_strategy=CollectionCheckStrategy.ALL_ITEMS)
def some_function(a: int, b: float, c: str, *args: str) -> bool:
    ...
    return retval
```

This also allows you to override the global options for specific functions when using the import hook.

Note: You should always place this decorator closest to the original function, as it will not work when there is another decorator wrapping the function. For the same reason, when you use it on a class that has wrapping decorators on its methods, such methods will not be instrumented. In contrast, the import hook has no such restrictions.

1.1.3 Using the import hook

The import hook, when active, automatically instruments all type annotated functions to type check arguments, return values and values yielded by or sent to generator functions. This allows for a non-invasive method of run time type checking. This method does not modify the source code on disk, but instead modifies its AST (Abstract Syntax Tree) when the module is loaded.

Using the import hook is as straightforward as installing it before you import any modules you wish to be type checked. Give it the name of your top level package (or a list of package names):

```
from typeguard import install_import_hook
```

(continues on next page)

(continued from previous page)

```
install_import_hook('myapp')
from myapp import some_module  # import only AFTER installing the hook, or it won't take␣
↪effect
```

If you wish, you can uninstall the import hook:

```
manager = install_import_hook('myapp')
from myapp import some_module
manager.uninstall()
```

or using the context manager approach:

```
with install_import_hook('myapp'):
    from myapp import some_module
```

You can also customize the logic used to select which modules to instrument:

```
from typeguard import TypeguardFinder, install_import_hook

class CustomFinder(TypeguardFinder):
    def should_instrument(self, module_name: str):
        # disregard the module names list and instrument all loaded modules
        return True

install_import_hook('', cls=CustomFinder)
```

1.1.4 Notes on forward reference handling

The internal type checking functions, injected to instrumented code by either *@typechecked* or the import hook, use the "naked" versions of any annotations, undoing any quotations in them (and the effects of `from __future__ import annotations`). As such, in instrumented code, the *forward_ref_policy* only applies when using type variables containing forward references, or type aliases likewise containing forward references.

To facilitate the use of types only available to static type checkers, Typeguard recognizes module-level imports guarded by `if typing.TYPE_CHECKING:` or `if TYPE_CHECKING:` (add the appropriate `typing` imports). Imports made within such blocks on the module level will be replaced in calls to internal type checking functions with `Any`.

1.1.5 Using the pytest plugin

Typeguard comes with a pytest plugin that installs the import hook (explained in the previous section). To use it, run `pytest` with the appropriate `--typeguard-packages` option. For example, if you wanted to instrument the `foo.bar` and `xyz` packages for type checking, you can do the following:

```
pytest --typeguard-packages=foo.bar,xyz
```

There is currently no support for specifying a customized module finder.

1.1.6 Setting configuration options

There are several configuration options that can be set that influence how type checking is done. The *typeguard.config* (which is of type *TypeCheckConfiguration*) controls the options applied to code instrumented via either *@typechecked* or the import hook. The *check_type()*, function, however, uses the built-in defaults and is not affected by the global configuration, so you must pass any configuration overrides explicitly with each call.

You can also override specific configuration options in instrumented functions (or entire classes) by passing keyword arguments to *@typechecked*. You can do this even if you're using the import hook, as the import hook will remove the decorator to ensure that no double instrumentation takes place. If you're using the import hook to type check your code only during tests and don't want to include `typeguard` as a run-time dependency, you can use a dummy replacement for the decorator.

For example, the following snippet will only import the decorator during a pytest run:

```
import sys

if "pytest" in sys.modules:
    from typeguard import typechecked
else:
    from typing import TypeVar
    _T = TypeVar("_T")

    def typechecked(target: _T, **kwargs) -> _T:
        return target if target else typechecked
```

1.1.7 Suppressing type checks

Temporarily disabling type checks

If you need to temporarily suppress type checking, you can use the *suppress_type_checks()* function, either as a context manager or a decorator, to skip the checks:

```
from typeguard import check_type, suppress_type_checks

with suppress_type_checks():
    check_type(1, str)  # would fail without the suppression

@suppress_type_checks
def my_suppressed_function(x: int) -> None:
    ...
```

Suppression state is tracked globally. Suppression ends only when all the context managers have exited and all calls to decorated functions have returned.

Permanently suppressing type checks for selected functions

To exclude specific functions from run time type checking, you can use one of the following decorators:

- *@typeguard_ignore*: prevents the decorated function from being instrumentated by the import hook
- `@no_type_check`: as above, but disables static type checking too

For example, calling the function defined below will not result in a type check error when the containing module is instrumented by the import hook:

```
from typeguard import typeguard_ignore

@typeguard_ignore
def f(x: int) -> int:
    return str(x)
```

Warning: The `@no_type_check_decorator` decorator is not currently recognized by Typeguard.

1.1.8 Suppressing the `@typechecked` decorator in production

If you're using the *@typechecked* decorator to gradually introduce run-time type checks to your code base, you can disable the checks in production by running Python in optimized mode (as opposed to debug mode which is the default mode). You can do this by either starting Python with the `-O` or `-OO` option, or by setting the PYTHONOPTIMIZE environment variable. This will cause *@typechecked* to become a no-op when the import hook is not being used to instrument the code.

1.1.9 Debugging instrumented code

If you find that your code behaves in an unexpected fashion with the Typeguard instrumentation in place, you should set the `typeguard.config.debug_instrumentation` flag to `True`. This will print all the instrumented code after the modifications, which you can check to find the reason for the unexpected behavior.

If you're using the pytest plugin, you can also pass the `--typeguard-debug-instrumentation` and `-s` flags together for the same effect.

1.2 Features

1.2.1 What does Typeguard check?

The following type checks are implemented in Typeguard:

- Types of arguments passed to instrumented functions
- Types of values returned from instrumented functions
- Types of values yielded from instrumented generator functions
- Types of values sent to instrumented generator functions
- Types of values assigned to local variables within instrumented functions

1.2.2 What does Typeguard NOT check?

The following type checks are not yet supported in Typeguard:

- Types of values assigned to class or instance variables
- Types of values assigned to global or nonlocal variables
- Stubs defined with `@overload` (the implementation is checked if instrumented)
- `yield_from` statements in generator functions
- `ParamSpec` and `Concatenate` are currently ignored
- Types where they are shadowed by arguments with the same name (e.g. `def foo(x: type, type: str): ...`)

1.2.3 Other limitations

Local references to nested classes

Forward references from methods pointing to non-local nested classes cannot currently be resolved:

```
class Outer:
    class Inner:
        pass

    # Cannot be resolved as the name is no longer available
    def method(self) -> "Inner":
        return Outer.Inner()
```

This shortcoming may be resolved in a future release.

Using `@typechecked` on top of other decorators

As *`@typechecked`* works by recompiling the target function with instrumentation added, it needs to replace all the references to the original function with the new one. This could be impossible when it's placed on top of another decorator that wraps the original function. It has no way of telling that other decorator that the target function should be switched to a new one. To work around this limitation, either place *`@typechecked`* at the bottom of the decorator stack, or use the import hook instead.

1.2.4 Special considerations for `if TYPE_CHECKING:`

Both the import hook and *`@typechecked`* avoid checking against anything imported in a module-level `if TYPE_CHECKING:` (or `if typing.TYPE_CHECKING:`) block, since those types will not be available at run time. Therefore, no errors or warnings are emitted for such annotations, even when they would normally not be found.

1.2.5 Support for generator functions

For generator functions, the checks applied depend on the function's return annotation. For example, the following function gets its yield, send and return values type checked:

```
from collections.abc import Generator

def my_generator() -> Generator[int, str, bool]:
    a = yield 6
    return True
```

In contrast, the following generator function only gets its yield value checked:

```
from collections.abc import Iterator

def my_generator() -> Iterator[int]:
    a = yield 6
    return True
```

Asynchronous generators work just the same way, except they don't support returning values other than `None`, so the annotation only has two items:

```
from collections.abc import AsyncGenerator

async def my_generator() -> AsyncGenerator[int, str]:
    a = yield 6
```

Overall, the following type annotations will work for generator function type checking:

- `typing.Generator`
- `collections.abc.Generator`
- `typing.Iterator`
- `collections.abc.Iterator`
- `typing.Iterable`
- `collections.abc.Iterable`
- `typing.AsyncIterator`
- `collections.abc.AsyncIterator`
- `typing.AsyncIterable`
- `collections.abc.AsyncIterable`
- `typing.AsyncGenerator`
- `collections.abc.AsyncGenerator`

1.2.6 Support for PEP 604 unions on Pythons older than 3.10

The PEP 604 `X | Y` notation was introduced in Python 3.10, but it can be used with older Python versions in modules where `from __future__ import annotations` is present. Typeguard contains a special parser that lets it convert these to older `Union` annotations internally.

1.2.7 Support for generic built-in collection types on Pythons older than 3.9

The built-in collection types (`list`, `tuple`, `dict`, `set` and `frozenset`) gained support for generics in Python 3.9. For earlier Python versions, Typeguard provides a way to work with such annotations by substituting them with the equivalent `typing` types. The only requirement for this to work is the use of `from __future__ import annotations` in all such modules.

1.2.8 Support for mock objects

Typeguard handles the `unittest.mock.Mock` class (and its subclasses) specially, bypassing any type checks when encountering instances of these classes. Note that any "spec" class passed to the mock object is currently not respected.

1.2.9 Supported standard library annotations

The following types from the standard library have specialized support:

Type(s)	Notes
`typing.Any`	Any type passes type checks against this annotation. Inheriting from `Any` (`typing.Any` on Python 3.11+, or `typing.extensions.Any`) will pass any type check
`typing.Annotated`	Original annotation is unwrapped and typechecked normally
`BinaryIO`	Specialized instance checks are performed
`typing.Callable` `collections.abc.Callable`	Argument count is checked but types are not (yet)
`dict` `typing.Dict`	Keys and values are typechecked
`typing.IO`	Specialized instance checks are performed
`list` `typing.List`	Contents are typechecked
`typing.Literal`	
`typing.LiteralString`	Checked as `str`

`typing.Mapping`
`typing.MutableMapping`
`collections.abc.Mapping`
`collections.abc.MutableMapping`
	Keys and values are typechecked

`typing.NamedTuple`	Field values are typechecked
`typing.Never` `typing.NoReturn`	Supported in argument and return type annotations
`typing.Protocol`	Run-time protocols are checked with `isinstance()`, others are ignored
`typing.Self`	

`set`
`frozenset`
`typing.Set`
`typing.AbstractSet`
	Contents are typechecked

`typing.Sequence` `collections.abc.Sequence`	Contents are typechecked
`typing.TextIO`	Specialized instance checks are performed
`tuple` `typing.Tuple`	Contents are typechecked
`type`	

1.3 Extending Typeguard

1.3.1 Adding new type checkers

The range of types supported by Typeguard can be extended by writing a **type checker lookup funvtion** and one or more **type checker functions**. The former will return one of the latter, or `None` if the given value does not match any of your custom type checker functions.

The lookup function receives three arguments:

1. The origin type (the annotation with any arguments stripped from it)
2. The previously stripped out generic arguments, if any
3. Extra arguments from the `Annotated` annotation, if any

For example, if the annotation was `tuple`,, the lookup function would be called with `tuple, (), ()`. If the type was parametrized, like `tuple[str, int]`, it would be called with `tuple, (str, int), ()`. If the annotation was `Annotated[tuple[str, int], "foo", "bar"]`, the arguments would instead be `tuple, (str, int), ("foo", "bar")`.

The checker function receives four arguments:

1. The value to be type checked
2. The origin type
3. The generic arguments from the annotation (empty tuple when the annotation was not parametrized)
4. The memo object (*TypeCheckMemo*)

There are a couple of things to take into account when writing a type checker:

1. If your type checker function needs to do further type checks (such as type checking items in a collection), you need to use *check_type_internal()* (and pass along `memo` to it)
2. If you're type checking collections, your checker function should respect the *collection_check_strategy* setting, available from *config*

Changed in version 4.0: In Typeguard 4.0, checker functions **must** respect the settings in `memo.config`, rather than the global configuration

The following example contains a lookup function and type checker for a custom class (`MySpecialType`):

```
from __future__ import annotations
from inspect import isclass
from typing import Any

from typeguard import TypeCheckError, TypeCheckerCallable, TypeCheckMemo


class MySpecialType:
    pass


def check_my_special_type(
    value: Any, origin_type: Any, args: tuple[Any, ...], memo: TypeCheckMemo
) -> None:
    if not isinstance(value, MySpecialType):
        raise TypeCheckError('is not my special type')
```

(continues on next page)

(continued from previous page)

```
def my_checker_lookup(
    origin_type: Any, args: tuple[Any, ...], extras: tuple[Any, ...]
) -> TypeCheckerCallable | None:
    if isclass(origin_type) and issubclass(origin_type, MySpecialType):
        return check_my_special_type

    return None
```

1.3.2 Registering your type checker lookup function with Typeguard

Just writing a type checker lookup function doesn't do anything by itself. You'll have to advertise your type checker lookup function to Typeguard somehow. There are two ways to do that (pick just one):

1. Append to *typeguard.checker_lookup_functions*
2. Add an entry point to your project in the `typeguard.checker_lookup` group

If you're packaging your project with standard packaging tools, it may be better to add an entry point instead of registering it manually, because manual registration requires the registration code to run first before the lookup function can work.

To manually register the type checker lookup function with Typeguard:

```
from typeguard import checker_lookup_functions

checker_lookup_functions.append(my_checker_lookup)
```

For adding entry points to your project packaging metadata, the exact method may vary depending on your packaging tool of choice, but the standard way (supported at least by recent versions of `setuptools`) is to add this to `pyproject.toml`:

```
[project.entry-points]
typeguard.checker_lookup = {myplugin = "myapp.my_plugin_module:my_checker_lookup"}
```

The configuration above assumes that the **globally unique** (within the `typeguard.checker_lookup` namespace) entry point name for your lookup function is `myplugin`, it lives in the `myapp.my_plugin_module` and the name of the function there is `my_checker_lookup`.

Note: After modifying your project configuration, you may have to reinstall it in order for the entry point to become discoverable.

1.4 Contributing to Typeguard

If you wish to contribute a fix or feature to Typeguard, please follow the following guidelines.

When you make a pull request against the main Typeguard codebase, Github runs the test suite against your modified code. Before making a pull request, you should ensure that the modified code passes tests and code quality checks locally.

1.4.1 Running the test suite

You can run the test suite two ways: either with tox, or by running pytest directly.

To run tox against all supported (of those present on your system) Python versions:

```
tox
```

Tox will handle the installation of dependencies in separate virtual environments.

To pass arguments to the underlying pytest command, you can add them after `--`, like this:

```
tox -- -k somekeyword
```

To use pytest directly, you can set up a virtual environment and install the project in development mode along with its test dependencies (virtualenv activation demonstrated for Linux and macOS; on Windows you need `venv\Scripts\activate` instead):

```
python -m venv venv
source venv/bin/activate
pip install -e .[test]
```

Now you can just run pytest:

```
pytest
```

1.4.2 Building the documentation

To build the documentation, run `tox -e docs`. This will place the documentation in `build/sphinx/html` where you can open `index.html` to view the formatted documentation.

Typeguard uses ReadTheDocs to automatically build the documentation so the above procedure is only necessary if you are modifying the documentation and wish to check the results before committing.

Typeguard uses pre-commit to perform several code style/quality checks. It is recommended to activate pre-commit on your local clone of the repository (using `pre-commit install`) to ensure that your changes will pass the same checks on GitHub.

1.4.3 Making a pull request on Github

To get your changes merged to the main codebase, you need a Github account.

1. Fork the repository (if you don't have your own fork of it yet) by navigating to the main Typeguard repository and clicking on "Fork" near the top right corner.
2. Clone the forked repository to your local machine with `git clone git@github.com/yourusername/typeguard`.
3. Create a branch for your pull request, like `git checkout -b myfixname`
4. Make the desired changes to the code base.
5. Commit your changes locally. If your changes close an existing issue, add the text `Fixes #XXX.` or `Closes #XXX.` to the commit message (where XXX is the issue number).
6. Push the changeset(s) to your forked repository (`git push`)
7. Navigate to Pull requests page on the original repository (not your fork) and click "New pull request"
8. Click on the text "compare across forks".
9. Select your own fork as the head repository and then select the correct branch name.
10. Click on "Create pull request".

If you have trouble, consult the pull request making guide on opensource.com.

1.5 API reference

1.5.1 Type checking

typeguard.**check_type**(*value*, *expected_type*, *, *forward_ref_policy=ForwardRefPolicy.WARN*, *typecheck_fail_callback=None*, *collection_check_strategy=CollectionCheckStrategy.FIRST_ITEM*)

Ensure that `value` matches `expected_type`.

The types from the `typing` module do not support `isinstance()` or `issubclass()` so a number of type specific checks are required. This function knows which checker to call for which type.

This function wraps *check_type_internal()* in the following ways:

- Respects type checking suppression (*suppress_type_checks()*)
- Forms a *TypeCheckMemo* from the current stack frame
- Calls the configured type check fail callback if the check fails

Note that this function is independent of the globally shared configuration in *typeguard.config*. This means that usage within libraries is safe from being affected configuration changes made by other libraries or by the integrating application. Instead, configuration options have the same default values as their corresponding fields in *TypeCheckConfiguration*.

Parameters

- **value** (`object`) – value to be checked against `expected_type`
- **expected_type** (`Any`) – a class or generic type instance
- **forward_ref_policy** (*ForwardRefPolicy*) – see *TypeCheckConfiguration.forward_ref_policy*

- **typecheck_fail_callback** (Optional[Callable[[*TypeCheckError*, *TypeCheckMemo*], Any]]) – see :attr`TypeCheckConfiguration.typecheck_fail_callback`
- **collection_check_strategy** (*CollectionCheckStrategy*) – see *TypeCheckConfiguration.collection_check_strategy*

Return type
Any

Returns
value, unmodified

Raises
TypeCheckError – if there is a type mismatch

@typeguard.**typechecked**(*target=None*, ***, *forward_ref_policy=<unset>*, *typecheck_fail_callback=<unset>*, *collection_check_strategy=<unset>*, *debug_instrumentation=<unset>*)

Instrument the target function to perform run-time type checking.

This decorator recompiles the target function, injecting code to type check arguments, return values, yield values (excluding `yield from`) and assignments to annotated local variables.

This can also be used as a class decorator. This will instrument all type annotated methods, including `@classmethod`, `@staticmethod`, and `@property` decorated methods in the class.

Note: When Python is run in optimized mode (`-O` or `-OO`, this decorator is a no-op). This is a feature meant for selectively introducing type checking into a code base where the checks aren't meant to be run in production.

Parameters

- **target** (Optional[TypeVar(T_CallableOrType, bound= Callable[..., Any])]) – the function or class to enable type checking for
- **forward_ref_policy** (*ForwardRefPolicy* | *Unset*) – override for *TypeCheckConfiguration.forward_ref_policy*
- **typecheck_fail_callback** (Union[Callable[[*TypeCheckError*, *TypeCheckMemo*], Any], *Unset*]) – override for *TypeCheckConfiguration.typecheck_fail_callback*
- **collection_check_strategy** (*CollectionCheckStrategy* | *Unset*) – override for *TypeCheckConfiguration.collection_check_strategy*
- **debug_instrumentation** (bool | *Unset*) – override for *TypeCheckConfiguration.debug_instrumentation*

Return type
Any

1.5.2 Import hook

typeguard.**install_import_hook**(*packages=None*, ***, *cls=<class 'typeguard.TypeguardFinder'>*)

Install an import hook that instruments functions for automatic type checking.

This only affects modules loaded **after** this hook has been installed.

Parameters

- **packages** (Iterable[str] | None) – an iterable of package names to instrument, or None to instrument all packages

- **cls** (type[*TypeguardFinder*]) – a custom meta path finder class

Return type
ImportHookManager

Returns
a context manager that uninstalls the hook on exit (or when you call `.uninstall()`)

New in version 2.6.

class typeguard.**TypeguardFinder**(*packages*, *original_pathfinder*)

Wraps another path finder and instruments the module with *@typechecked* if *should_instrument()* returns `True`.

Should not be used directly, but rather via *install_import_hook()*.

New in version 2.6.

should_instrument(*module_name*)

Determine whether the module with the given name should be instrumented.

Parameters
module_name (`str`) – full name of the module that is about to be imported (e.g. `xyz.abc`)

Return type
`bool`

class typeguard.**ImportHookManager**(*hook*)

A handle that can be used to uninstall the Typeguard import hook.

uninstall()

Uninstall the import hook.

Return type
`None`

1.5.3 Configuration

typeguard.**config: *TypeCheckConfiguration***

The global configuration object.

Used by *@typechecked* and *install_import_hook()*, and notably **not used** by *check_type()*.

class typeguard.**TypeCheckConfiguration**(*forward_ref_policy=ForwardRefPolicy.WARN*, *typecheck_fail_callback=None*, *collection_check_strategy=CollectionCheckStrategy.FIRST_ITEM*, *debug_instrumentation=False*)

You can change Typeguard's behavior with these settings.

typecheck_fail_callback: Callable[[*TypeCheckError*, *TypeCheckMemo*], Any]

Callable that is called when type checking fails.

Default: `None` (the *TypeCheckError* is raised directly)

forward_ref_policy: *ForwardRefPolicy*

Specifies what to do when a forward reference fails to resolve.

Default: `WARN`

collection_check_strategy: *CollectionCheckStrategy*

Specifies how thoroughly the contents of collections (list, dict, etc.) are type checked.

Default: `FIRST_ITEM`

debug_instrumentation: **bool**

If set to `True`, the code of modules or functions instrumented by typeguard is printed to `sys.stderr` after the instrumentation is done

Requires Python 3.9 or newer.

Default: `False`

class typeguard.**CollectionCheckStrategy**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Specifies how thoroughly the contents of collections are type checked.

This has an effect on the following built-in checkers:

- `AbstractSet`
- `Dict`
- `List`
- `Mapping`
- `Set`
- `Tuple[<type>, ...]` (arbitrarily sized tuples)

Members:

- `FIRST_ITEM`: check only the first item
- `ALL_ITEMS`: check all items

class typeguard.**Unset**

class typeguard.**ForwardRefPolicy**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Defines how unresolved forward references are handled.

Members:

- `ERROR`: propagate the `NameError` when the forward reference lookup fails
- `WARN`: emit a *TypeHintWarning* if the forward reference lookup fails
- `IGNORE`: silently skip checks for unresolveable forward references

typeguard.**warn_on_error**(*exc*, *memo*)

Emit a warning on a type mismatch.

This is intended to be used as an error handler in *TypeCheckConfiguration.typecheck_fail_callback*.

Return type
None

1.5.4 Custom checkers

typeguard.**check_type_internal**(*value*, *annotation*, *memo*)

Check that the given object is compatible with the given type annotation.

This function should only be used by type checker callables. Applications should use *check_type()* instead.

Parameters

- **value** (Any) – the value to check
- **annotation** (Any) – the type annotation to check against
- **memo** (*TypeCheckMemo*) – a memo object containing configuration and information necessary for looking up forward references

Return type
None

typeguard.**load_plugins**()

Load all type checker lookup functions from entry points.

All entry points from the `typeguard.checker_lookup` group are loaded, and the returned lookup functions are added to *typeguard.checker_lookup_functions*.

Note: This function is called implicitly on import, unless the `TYPEGUARD_DISABLE_PLUGIN_AUTOLOAD` environment variable is present.

Return type
None

typeguard.**checker_lookup_functions: list[Callable[[Any, Tuple[Any, ...], Tuple[Any, ...]], Callable[[Any, Any, Tuple[Any, ...], *TypeCheckMemo*], Any] | None]]**

A list of callables that are used to look up a checker callable for an annotation.

class typeguard.**TypeCheckMemo**(*globals*, *locals*, *, *self_type=None*, *config=TypeCheckConfiguration(forward_ref_policy=<ForwardRefPolicy.WARN: 2>*, *typecheck_fail_callback=None*, *collection_check_strategy=<CollectionCheckStrategy.FIRST_ITEM: 1>*, *debug_instrumentation=False)*)

Contains information necessary for type checkers to do their work.

globals: dict[str, Any]

Dictionary of global variables to use for resolving forward references.

locals: dict[str, Any]

Dictionary of local variables to use for resolving forward references.

self_type: type | None

When running type checks within an instance method or class method, this is the class object that the first argument (usually named `self` or `cls`) refers to.

config: *TypeCheckConfiguration*

Contains the configuration for a particular set of type checking operations.

1.5.5 Type check suppression

@typeguard.**typeguard_ignore**

Decorator to indicate that annotations are not type hints.

The argument must be a class or function; if it is a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses).

This mutates the function(s) or class(es) in place.

typeguard.**suppress_type_checks**(*func=None*)

Temporarily suppress all type checking.

This function has two operating modes, based on how it's used:

1. as a context manager (`with suppress_type_checks(): ...`)
2. as a decorator (`@suppress_type_checks`)

When used as a context manager, *check_type()* and any automatically instrumented functions skip the actual type checking. These context managers can be nested.

When used as a decorator, all type checking is suppressed while the function is running.

Type checking will resume once no more context managers are active and no decorated functions are running.

Both operating modes are thread-safe.

Return type
`Union[Callable[ParamSpec, TypeVar(T)], ContextManager[None]]`

1.5.6 Exceptions and warnings

exception typeguard.**InstrumentationWarning**(*message*)

Emitted when there's a problem with instrumenting a function for type checks.

exception typeguard.**TypeCheckError**(*message*)

Raised by typeguard's type checkers when a type mismatch is detected.

exception typeguard.**TypeCheckWarning**(*message*)

Emitted by typeguard's type checkers when a type mismatch is detected.

exception typeguard.**TypeHintWarning**

A warning that is emitted when a type hint in string form could not be resolved to an actual type.

1.6 Version history

This library adheres to Semantic Versioning 2.0.

4.0.0 (2023-05-12)

- No changes

4.0.0rc6 (2023-05-07)

- Fixed `@typechecked` optimization causing compilation of instrumented code to fail when an `if` block was left empty by the AST transformer (#352)

- Fixed the AST transformer trying to parse the second argument of `typing.Annotated` as a forward reference (#353)

4.0.0rc5 (2023-05-01)

- Added `InstrumentationWarning` to the public API
- Changed `@typechecked` to skip instrumentation in optimized mode, as in typeguard 2.x
- Avoid type checks where the types in question are shadowed by local variables
- Fixed instrumentation using `typing.Optional` without a subscript when the subscript value was erased due to being an ignored import
- Fixed `TypeError: isinstance() arg 2 must be a type or tuple of types` when instrumented code tries to check a value against a naked (`str`, not `ForwardRef`) forward reference
- Fixed instrumentation using the wrong "self" type in the `__new__()` method

4.0.0rc4 (2023-04-15)

- Fixed imports guarded by `if TYPE_CHECKING:` when used with subscripts (`SomeType[...]`) being replaced with `Any[...]` instead of just `Any`
- Fixed instrumentation inadvertently mutating a function's annotations on Python 3.7 and 3.8
- Fixed `Concatenate[...]` in `Callable` parameters causing `TypeError` to be raised
- Fixed type checks for `*args` or `**kwargs` not being suppressed when their types are unusable (guarded by `if TYPE_CHECKING:` or otherwise)
- Fixed `TypeError` when checking against a generic `NewType`
- Don't try to check types shadowed by argument names (e.g. `def foo(x: type, type: str): ...`)
- Don't check against unions where one of the elements is `Any`

4.0.0rc3 (2023-04-10)

- Fixed `typing.Literal` subscript contents being evaluated as forward references
- Fixed resolution of forward references in type aliases

4.0.0rc2 (2023-04-08)

- The `.pyc` files now use a version-based optimization suffix in the file names so as not to cause the interpreter to load potentially faulty/incompatible cached bytecode generated by older versions
- Fixed typed variable positional and keyword arguments causing compilation errors on Python 3.7 and 3.8
- Fixed compilation error when a type annotation contains a type guarded by `if TYPE_CHECKING:`

4.0.0rc1 (2023-04-02)

- **BACKWARD INCOMPATIBLE** `check_type()` no longer uses the global configuration. It now uses the default configuration values, unless overridden with an explicit `config` argument.
- **BACKWARD INCOMPATIBLE** Removed `CallMemo` from the API
- **BACKWARD INCOMPATIBLE** Required checkers to use the configuration from `memo.config`, rather than the global configuration
- Added keyword arguments to `@typechecked`, allowing users to override settings on a per-function basis
- Added support for using `suppress_type_checks()` as a decorator
- Added support for type checking against nonlocal classes defined within the same parent function as the instrumented function

- Changed instrumentation to statically copy the function annotations to avoid having to look up the function object at run time
- Improved support for avoiding type checks against imports declared in `if TYPE_CHECKING:` blocks
- Fixed `check_type` not returning the passed value when checking against `Any`, or when type checking is being suppressed
- Fixed `suppress_type_checks()` not ending the suppression if the context block raises an exception
- Fixed checking non-dictionary objects against a `TypedDict` annotation (PR by Tolker-KU)

3.0.2 (2023-03-22)

- Improved warnings by ensuring that they target user code and not Typeguard internal code
- Fixed `warn_on_error()` not showing where the type violation actually occurred
- Fixed local assignment to `*args` or `**kwargs` being type checked incorrectly
- Fixed `TypeError` on `check_type(..., None)`
- Fixed unpacking assignment not working with a starred variable (`x, *y = ...`) in the target tuple
- Fixed variable multi-assignment (`a = b = c = ...`) being type checked incorrectly

3.0.1 (2023-03-16)

- Improved the documentation
- Fixed assignment unpacking (`a, b = ...`) being checked incorrectly
- Fixed `@typechecked` attempting to instrument wrapper decorators such as `@contextmanager` when applied to a class
- Fixed `py.typed` missing from the wheel when not building from a git checkout

3.0.0 (2023-03-15)

- **BACKWARD INCOMPATIBLE** Dropped the `argname`, `memo`, `globals` and `locals` arguments from `check_type()`
- **BACKWARD INCOMPATIBLE** Removed the `check_argument_types()` and `check_return_type()` functions (use `@typechecked` instead)
- **BACKWARD INCOMPATIBLE** Moved `install_import_hook` to be directly importable from the `typeguard` module
- **BACKWARD INCOMPATIBLE** Changed the checking of collections (list, set, dict, sequence, mapping) to only check the first item by default. To get the old behavior, set `typeguard.config.collection_check_strategy` to `CollectionCheckStrategy.ALL_ITEMS`
- **BACKWARD INCOMPATIBLE** Type checking failures now raise `typeguard.TypeCheckError` instead of `TypeError`
- Dropped Python 3.5 and 3.6 support
- Dropped the deprecated profiler hook (`TypeChecker`)
- Added a configuration system
- Added support for custom type checking functions
- Added support for PEP 604 union types (`X | Y`) on all Python versions
- Added support for generic built-in collection types (`list[int]` et al) on all Python versions
- Added support for checking arbitrary `Mapping` types

- Added support for the `Self` type
- Added support for `typing.Never` (and `typing_extensions.Never`)
- Added support for `Never` and `NoReturn` in argument annotations
- Added support for `LiteralString`
- Added support for `TypeGuard`
- Added support for the subclassable `Any` on Python 3.11 and `typing_extensions`
- Added the possibility to have the import hook instrument all packages
- Added the `suppress_type_checks()` context manager function for temporarily disabling type checks
- Much improved error messages showing where the type check failed
- Made it possible to apply `@typechecked` on top of `@classmethod` / `@staticmethod` (PR by jacobpbrugh)
- Changed `check_type()` to return the passed value, so it can be used (to an extent) in place of `typing.cast()`, but with run-time type checking
- Replaced custom implementation of `is_typeddict()` with the implementation from `typing_extensions` v4.1.0
- Emit `InstrumentationWarning` instead of raising `RuntimeError` from the pytest plugin if modules in the target package have already been imported
- Fixed `TypeError` when checking against `TypedDict` when the value has mixed types among the extra keys (PR by biolds)
- Fixed incompatibility with `typing_extensions` v4.1+ on Python 3.10 (PR by David C.)
- Fixed checking of `Tuple[()]` on Python 3.11 and `tuple[()]` on Python 3.9+
- Fixed integers 0 and 1 passing for `Literal[False]` and `Literal[True]`, respectively
- Fixed type checking of annotated variable positional and keyword arguments (`*args` and `**kwargs`)
- Fixed checks against `unittest.Mock` and derivatives being done in the wrong place

2.13.3 (2021-12-10)

- Fixed `TypeError` when using typeguard within `exec()` (where `__module__` is `None`) (PR by Andy Jones)
- Fixed `TypedDict` causing `TypeError: TypedDict does not support instance and class checks` on Python 3.8 with standard library (not `typing_extensions`) typed dicts

2.13.2 (2021-11-23)

- Fixed `typing_extensions` being imported unconditionally on Python < 3.9 (bug introduced in 2.13.1)

2.13.1 (2021-11-23)

- Fixed `@typechecked` replacing abstract properties with regular properties
- Fixed any generic type subclassing `Dict` being mistakenly checked as `TypedDict` on Python 3.10

2.13.0 (2021-10-11)

- Added support for returning `NotImplemented` from binary magic methods (`__eq__()` et al)
- Added support for checking union types (e.g. `Type[Union[X, Y]]`)
- Fixed error message when a check against a `Literal` fails in a union on Python 3.10
- Fixed `NewType` not being checked on Python 3.10
- Fixed unwarranted warning when `@typechecked` is applied to a class that contains unannotated properties

- Fixed `TypeError` in the async generator wrapper due to changes in `__aiter__()` protocol
- Fixed broken `TypeVar` checks – variance is now (correctly) disregarded, and only bound types and constraints are checked against (but type variable resolution is not done)

2.12.1 (2021-06-04)

- Fixed `AttributeError` when `__code__` is missing from the checked callable (PR by epenet)

2.12.0 (2021-04-01)

- Added `@typeguard_ignore` decorator to exclude specific functions and classes from runtime type checking (PR by Claudio Jolowicz)

2.11.1 (2021-02-16)

- Fixed compatibility with Python 3.10

2.11.0 (2021-02-13)

- Added support for type checking class properties (PR by Ethan Pronovost)
- Fixed static type checking of `@typechecked` decorators (PR by Kenny Stauffer)
- Fixed wrong error message when type check against a `bytes` declaration fails
- Allowed `memoryview` objects to pass as `bytes` (like MyPy does)
- Shortened tracebacks (PR by prescod)

2.10.0 (2020-10-17)

- Added support for Python 3.9 (PR by Csergő Bálint)
- Added support for nested `Literal`
- Added support for `TypedDict` inheritance (with some caveats; see the user guide on that for details)
- An appropriate `TypeError` is now raised when encountering an illegal `Literal` value
- Fixed checking `NoReturn` on Python < 3.8 when `typing_extensions` was not installed
- Fixed import hook matching unwanted modules (PR by Wouter Bolsterlee)
- Install the pytest plugin earlier in the test run to support more use cases (PR by Wouter Bolsterlee)

2.9.1 (2020-06-07)

- Fixed `ImportError` on Python < 3.8 when `typing_extensions` was not installed

2.9.0 (2020-06-06)

- Upped the minimum Python version from 3.5.2 to 3.5.3
- Added support for `typing.NoReturn`
- Added full support for `typing_extensions` (now equivalent to support of the `typing` module)
- Added the option of supplying `check_type()` with globals/locals for correct resolution of forward references
- Fixed erroneous `TypeError` when trying to check against non-runtime `typing.Protocol` (skips the check for now until a proper compatibility check has been implemented)
- Fixed forward references in `TypedDict` not being resolved
- Fixed checking against recursive types

2.8.0 (2020-06-02)

- Added support for the `Mock` and `MagicMock` types (PR by prescod)

- Added support for `typing_extensions.Literal` (PR by Ryan Rowe)
- Fixed unintended wrapping of untyped generators (PR by prescod)
- Fixed checking against bound type variables with `check_type()` without a call memo
- Fixed error message when checking against a `Union` containing a `Literal`

2.7.1 (2019-12-27)

- Fixed `@typechecked` returning `None` when called with `always=True` and Python runs in optimized mode
- Fixed performance regression introduced in v2.7.0 (the `getattr_static()` call was causing a 3x slowdown)

2.7.0 (2019-12-10)

- Added support for `typing.Protocol` subclasses
- Added support for `typing.AbstractSet`
- Fixed the handling of `total=False` in `TypedDict`
- Fixed no error reported on unknown keys with `TypedDict`
- Removed support of default values in `TypedDict`, as they are not supported in the spec

2.6.1 (2019-11-17)

- Fixed import errors when using the import hook and trying to import a module that has both a module docstring and `__future__` imports in it
- Fixed `AttributeError` when using `@typechecked` on a metaclass
- Fixed `@typechecked` compatibility with built-in function wrappers
- Fixed type checking generator wrappers not being recognized as generators
- Fixed resolution of forward references in certain cases (inner classes, function-local classes)
- Fixed `AttributeError` when a class has contains a variable that is an instance of a class that has a `__call__()` method
- Fixed class methods and static methods being wrapped incorrectly when `@typechecked` is applied to the class
- Fixed `AttributeError` when `@typechecked` is applied to a function that has been decorated with a decorator that does not properly wrap the original (PR by Joel Beach)
- Fixed collections with mixed value (or key) types raising `TypeError` on Python 3.7+ when matched against unparametrized annotations from the `typing` module
- Fixed inadvertent `TypeError` when checking against a type variable that has constraints or a bound type expressed as a forward reference

2.6.0 (2019-11-06)

- Added a **PEP 302** import hook for annotating functions and classes with `@typechecked`
- Added a pytest plugin that activates the import hook
- Added support for `typing.TypedDict`
- Deprecated `TypeChecker` (will be removed in v3.0)

2.5.1 (2019-09-26)

- Fixed incompatibility between annotated `Iterable`, `Iterator`, `AsyncIterable` or `AsyncIterator` return types and generator/async generator functions
- Fixed `TypeError` being wrapped inside another TypeError (PR by russok)

2.5.0 (2019-08-26)

- Added yield type checking via `TypeChecker` for regular generators
- Added yield, send and return type checking via `@typechecked` for regular and async generators
- Silenced `TypeChecker` warnings about async generators
- Fixed bogus `TypeError` on `Type[Any]`
- Fixed bogus `TypeChecker` warnings when an exception is raised from a type checked function
- Accept a `bytearray` where `bytes` are expected, as per python/typing#552
- Added policies for dealing with unmatched forward references
- Added support for using `@typechecked` as a class decorator
- Added `check_return_type()` to accompany `check_argument_types()`
- Added Sphinx documentation

2.4.1 (2019-07-15)

- Fixed broken packaging configuration

2.4.0 (2019-07-14)

- Added **PEP 561** support
- Added support for empty tuples (`Tuple[()]`)
- Added support for `typing.Literal`
- Make getting the caller frame faster (PR by Nick Sweeting)

2.3.1 (2019-04-12)

- Fixed thread safety issue with the type hints cache (PR by Kelsey Francis)

2.3.0 (2019-03-27)

- Added support for `typing.IO` and derivatives
- Fixed return type checking for coroutine functions
- Dropped support for Python 3.4

2.2.2 (2018-08-13)

- Fixed false positive when checking a callable against the plain `typing.Callable` on Python 3.7

2.2.1 (2018-08-12)

- Argument type annotations are no longer unioned with the types of their default values, except in the case of `None` as the default value (although PEP 484 still recommends against this)
- Fixed some generic types (`typing.Collection` among others) producing false negatives on Python 3.7
- Shortened unnecessarily long tracebacks by raising a new `TypeError` based on the old one
- Allowed type checking against arbitrary types by removing the requirement to supply a call memo to `check_type()`
- Fixed `AttributeError` when running with the pydev debugger extension installed
- Fixed getting type names on `typing.*` on Python 3.7 (fix by Dale Jung)

2.2.0 (2018-07-08)

- Fixed compatibility with Python 3.7

- Removed support for Python 3.3
- Added support for `typing.NewType` (contributed by reinhrst)

2.1.4 (2018-01-07)

- Removed support for backports.typing, as it has been removed from PyPI
- Fixed checking of the numeric tower (complex -> float -> int) according to PEP 484

2.1.3 (2017-03-13)

- Fixed type checks against generic classes

2.1.2 (2017-03-12)

- Fixed leak of function objects (should've used a `WeakValueDictionary` instead of `WeakKeyDictionary`)
- Fixed obscure failure of TypeChecker when it's unable to find the function object
- Fixed parametrized `Type` not working with type variables
- Fixed type checks against variable positional and keyword arguments

2.1.1 (2016-12-20)

- Fixed formatting of README.rst so it renders properly on PyPI

2.1.0 (2016-12-17)

- Added support for `typings.Type` (available in Python 3.5.2+)
- Added a third, `sys.setprofile()` based type checking approach (`typeguard.TypeChecker`)
- Changed certain type error messages to display "function" instead of the function's qualified name

2.0.2 (2016-12-17)

- More Python 3.6 compatibility fixes (along with a broader test suite)

2.0.1 (2016-12-10)

- Fixed additional Python 3.6 compatibility issues

2.0.0 (2016-12-10)

- **BACKWARD INCOMPATIBLE** Dropped Python 3.2 support
- Fixed incompatibility with Python 3.6
- Use `inspect.signature()` in place of `inspect.getfullargspec`
- Added support for `typing.NamedTuple`

1.2.3 (2016-09-13)

- Fixed `@typechecked` skipping the check of return value type when the type annotation was `None`

1.2.2 (2016-08-23)

- Fixed checking of homogenous Tuple declarations (`Tuple[bool, ...]`)

1.2.1 (2016-06-29)

- Use `backports.typing` when possible to get new features on older Pythons
- Fixed incompatibility with Python 3.5.2

1.2.0 (2016-05-21)

- Fixed argument counting when a class is checked against a Callable specification

- Fixed argument counting when a functools.partial object is checked against a Callable specification
- Added checks against mandatory keyword-only arguments when checking against a Callable specification

1.1.3 (2016-05-09)

- Gracefully exit if `check_type_arguments` can't find a reference to the current function

1.1.2 (2016-05-08)

- Fixed TypeError when checking a builtin function against a parametrized Callable

1.1.1 (2016-01-03)

- Fixed improper argument counting with bound methods when typechecking callables

1.1.0 (2016-01-02)

- Eliminated the need to pass a reference to the currently executing function to `check_argument_types()`

1.0.2 (2016-01-02)

- Fixed types of default argument values not being considered as valid for the argument

1.0.1 (2016-01-01)

- Fixed type hints retrieval being done for the wrong callable in cases where the callable was wrapped with one or more decorators

1.0.0 (2015-12-28)

- Initial release

PYTHON MODULE INDEX

t

typeguard, 15

INDEX