Typeguard

Release 2.7.1.post3

Alex Grönholm

May 28, 2020

CONTENTS

CHAPTER

ONE

QUICK LINKS

1.1 API reference

1.2 typeguard

check_type(*argname*, *value*, *expected_type*, *memo=None*)
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.

Parameters

- **argname** (`str`) – name of the argument to check; used for error messages
- **value** – value to be checked against `expected_type`
- **expected_type** – a class or generic type instance

Return type `None`

check_argument_types(*memo=None*)
Check that the argument values match the annotated types.

Unless both `args` and `kwargs` are provided, the information will be retrieved from the previous stack frame (ie. from the function that called this).

Return type `bool`

Returns `True`

Raises **TypeError** – if there is an argument type mismatch

check_return_type(*retval*, *memo=None*)
Check that the return value is compatible with the return value annotation in the function.

Parameters **retval** – the value about to be returned from the call

Return type `bool`

Returns `True`

Raises **TypeError** – if there is a type mismatch

@typechecked(*func=None*, *, *always=False*, *_localns=None*)
Perform runtime type checking on the arguments that are passed to the wrapped function.

The return value is also checked against the return annotation if any.

If the __debug__ global variable is set to False, no wrapping and therefore no type checking is done, unless always is True.

This can also be used as a class decorator. This will wrap all type annotated methods in the class with this decorator.

Parameters

- **func** – the function or class to enable type checking for
- **always** – True to enable type checks even in optimized mode

class TypeChecker(*packages*, ***, *all_threads=True*, *forward_refs_policy=<ForwardRefPolicy.ERROR: 1>*)

A type checker that collects type violations by hooking into sys.setprofile().

Parameters

- **packages** (Union[str, Sequence[str]]) – list of top level modules and packages or modules to include for type checking
- **all_threads** (bool) – True to check types in all threads created while the checker is running, False to only check in the current one
- **forward_refs_policy** (*ForwardRefPolicy*) – how to handle unresolvable forward references in annotations

Deprecated since version 2.6: Use *install_import_hook()* instead. This class will be removed in v3.0.

exception TypeHintWarning

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

exception TypeWarning(*memo*, *event*, *frame*, *exception*)

A warning that is emitted when a type check fails.

Variables

- **event** (*str*) – call or return
- **func** (*Callable*) – the function in which the violation occurred (the called function if event is call, or the function where a value of the wrong type was returned from if event is return)
- **error** (*str*) – the error message contained by the caught TypeError
- **frame** – the frame in which the violation occurred

class ForwardRefPolicy

Defines how unresolved forward references are handled.

ERROR = 1

propagate the NameError from get_type_hints()

GUESS = 3

replace the annotation with the argument's class if the qualified name matches, else remove the annotation

WARN = 2

remove the annotation and emit a TypeHintWarning

1.3 typeguard.importhook

class 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`

install_import_hook(*packages*, *, *cls=<class 'typeguard.importhook.TypeguardFinder'>*)

Install an import hook that decorates classes and functions with `@typechecked`.

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

Return type `ImportHookManager`

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

New in version 2.6.

1.4 Version history

This library adheres to Semantic Versioning 2.0.

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

CHAPTER

TWO

USING TYPE CHECKER FUNCTIONS

Two functions are provided, potentially for use with the `assert` statement:

- *check_argument_types()*
- *check_return_type()*

These can be used to implement fine grained type checking for select functions. If the function is called with incompatible types, or *check_return_type()* is used and the return value does not match the return type annotation, then a `TypeError` is raised.

For example:

```
from typeguard import check_argument_types, check_return_value

def some_function(a: int, b: float, c: str, *args: str) -> bool:
    assert check_argument_types()
    ...
    assert check_return_value(retval)
    return retval
```

When combined with the `assert` statement, these checks are automatically removed from the code by the compiler when Python is executed in optimized mode (by passing the `-O` switch to the interpreter, or by setting the `PYTHONOPTIMIZE` environment variable to `1` (or higher).

Note: This method is not reliable when used in nested functions (i.e. functions defined inside other functions). This is because this operating mode relies on finding the correct function object using the garbage collector, and when a nested function is running, its function object may no longer be around anymore, as it is only bound to the closure of the enclosing function. For this reason, it is recommended to use `@typechecked` instead for nested functions.

CHAPTER

THREE

USING THE DECORATOR

The simplest way to type checking of both argument values and the return value for a single function is to use the `@typechecked` decorator:

```
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) are type␣
↪checked.
    # Does not apply to inner classes!
    def method(x: int) -> int:
        ...
```

The decorator works just like the two previously mentioned checker functions except that it has no issues with nested functions. The drawback, however, is that it adds one stack frame per wrapped function which may make debugging harder.

When a generator function is wrapped with `@typechecked`, the yields, sends and the return value are also type checked against the `Generator` annotation. The same applies to the yields and sends of an async generator (annotated with `AsyncGenerator`).

Note: The decorator also respects the optimized mode setting so it does nothing when the interpreter is running in optimized mode.

CHAPTER

FOUR

USING THE PROFILER HOOK

Deprecated since version 2.6: Use the import hook instead. The profiler hook will be removed in v3.0.

This type checking approach requires no code changes, but does come with a number of drawbacks. It relies on setting a profiler hook in the interpreter which gets called every time a new Python stack frame is entered or exited.

The easiest way to use this approach is to use a *TypeChecker* as a context manager:

```
from warnings import filterwarnings

from typeguard import TypeChecker, TypeWarning

# Display all TypeWarnings, not just the first one
filterwarnings('always', category=TypeWarning)

# Run your entire application inside this context block
with TypeChecker(['mypackage', 'otherpackage']):
    mypackage.run_app()
```

Alternatively, manually start (and stop) the checker:

```
checker = TypeChecker(['mypackage', 'otherpackage'])
checker.start()
mypackage.start_app()
```

The profiler hook approach has the following drawbacks:

- Return values of `None` are not type checked, as they cannot be distinguished from exceptions being raised
- The hook relies on finding the target function using the garbage collector which may make it miss some type violations, especially with nested functions
- Generator yield types are checked, send types are not
- Generator yields cannot be distinguished from returns
- Async generators are not type checked at all

Hint: Some other things you can do with *TypeChecker*:

- Display all warnings from the start with `python -W always::typeguard.TypeWarning`
- Redirect them to logging using `logging.captureWarnings()`
- Record warnings in your pytest test suite and fail test(s) if you get any (see the pytest documentation about that)

CHAPTER

FIVE

USING THE IMPORT HOOK

The import hook, when active, automatically decorates all type annotated functions with `@typechecked`. This allows for a noninvasive 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.importhook import install_import_hook

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.importhook 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)
```

CHAPTER

SIX

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.

CHAPTER

SEVEN

CHECKING TYPES DIRECTLY

Typeguard can also be used as a beefed-up version of `isinstance()` that also supports checking against annotations in the `typing` module:

```
from typeguard import check_type

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

CHAPTER

EIGHT

SUPPORTED TYPING.* TYPES

The following types from the `typing` package have specialized support:

Type	Notes
`AbstractSet`	Contents are typechecked
`Callable`	Argument count is checked but types are not (yet)
`Dict`	Keys and values are typechecked
`List`	Contents are typechecked
`Literal`	
`NamedTuple`	Field values are typechecked
`Protocol`	Value type checked with `issubclass()` against the protocol
`Set`	Contents are typechecked
`Sequence`	Contents are typechecked
`Tuple`	Contents are typechecked
`Type`	
`TypedDict`	Contents are typechecked
`TypeVar`	Constraints, bound types and co/contravariance are supported but custom generic types are not (due to type erasure)
`Union`	

PYTHON MODULE INDEX

t

INDEX