runtime_typing

Jonathan Scholbach

Feb 09, 2022

CONTENTS

runtime_typing is a python module for runtime validating function arguments and return values against constraints defined by type annotation using python's typing module.

The project is on *GitHub <https://github.com/jonathan-scholbach/runtime_typing>_*

The following types and *typing*-constructs are covered:

- primitives (python builtin-types, custom classes)
- *typing.Any*
- *typing.AnyStr*
- *typing.Callable*
- *typing.Dict*
- *typing.Iterable*
- *typing.Literal*
- *typing.Optional*
- *typing.Tuple*
- *typing.Type*
- *typing.TypedDict*
- *typing.TypeVar*
- *typing.Union*

CHAPTER

ONE

REQUIREMENTS AND INSTALLATION

1.1 Requirements

```
python>=3.8
```

1.2 Installation

```
pip install runtime_typing
```

CHAPTER

TWO

REFERENCE

2.1 @typed Decorator

runtime_typing.**typed**(*obj: Callable*, *mode: Literal['raise', 'warn', 'return'] = 'raise'*, *defer: bool = False*, *exclude: Optional[Iterable[str]] = None*, *include: Optional[Iterable[str]] = None*) → Callable

Decorator for validating arguments against type annotations.

Parameters

- **obj** – The object to be typed (either a function or a class). When a class is decorated, all its methods are typed, except for classmethods. Subclasses are not typed subsequently. See Examples below.
- **mode** – Mode how to handle typing violations. Default: *'raise'*
  - *'raise'*: For any violation of a type constraint, a *runtime_typing.RuntimeTypingError* is raised.
  - *'warn'*: For any violation of a type constraint, a *runtime_typing.RuntimeTypingWarning* is being thrown.
  - *'return'*: No exception is raised and no warning is thrown, but the return value of the function is a 2-Tuple, consisting of the original result of the function and a (possibly empty) list of *runtime_typing.TypingViolation*.
- **defer** – Whether to defer the handling of a violation. Default: *False*. By default, *@typed* handles every violation as soon as it occurs. This behavior can be changed by setting *defer* to *True*. This will gather all violations before handling them (i.e. throwing an Exception or a Warning)
- **include** – Iterable of names of arguments (can also contain "return") to be taken into account for type-checking. If falsey (an empty iterable, or not provided), all type-annotated arguments of the function are taken into account (except for those listed in the *exclude* parameter).
- **exclude** – Iterable of names of arguments (can also contatin "return") to be ignored during type-checking. Definitions via *exclude* prevail over those via *include*.

Example

Simple usage of the *@typed* decorator on a function.

```
@typed
def identity_of_int(x: int) -> int:
   return x
```

```
>>> identity_of_int("not an int")
RuntimeTypingError: TypingViolation in function `identity_of_int`: Expected type of␣
↪argument `x` to be `<class 'int'>` (got `<class 'str'>`).
```

Example

Usage with *typing* types.

```
from typing import Union

@typed
def identity_of_number(x: "Union[int, float]") -> "Union[int, float]":
    return x
```

```
>>> identity_of_number("not a number")
RuntimeTypingError: TypingViolation in function `identity_of_number`: Expected type␣
↪of argument `x` to be one of [<class 'int'>, <class 'float'>] (got `<class 'str'>
↪`).
```

Example

Make function return violations instead of raising with *mode="return"*.

```
@typed(mode="return")
def identity_of_int(x: int) -> int:
   return x
```

```
>>> identity_of_int("This does not raise.")
('This does not raise.', [TypingViolation in function `identity_of_int`: Expected␣
↪type of argument `x` to be `<class 'int'>` (got `<class 'str'>`).,␣
↪TypingViolation in function `identity_of_int`: Expected type of argument `return`␣
↪to be `<class 'int'>` (got `<class 'str'>`).])
```

Example

Defer raising violations with *defer=True*.

```
@typed(defer=True)
def identity_of_int(x: int) -> int:
    return x
```

```
>>> identity_of_int("not an int")
RuntimeTypingError:
    + TypingViolation in function `identity_of_int`: Expected type of argument `x`␣
↪to be `<class 'int'>` (got `<class 'str'>`).
    + TypingViolation in function `identity_of_int`: Expected type of argument␣
↪`return` to be `<class 'int'>` (got `<class 'str'>`).
```

Example

Use *include* and *exclude* parameters to restrict the function-arguments which are exposed to typechecking:

No Exception is raised in the following example, because only the return value is type-checked:

```
@typed(include=("return",))
def check_return_only(x: int) -> str:
    return str(x)
```

```
>>> check_only("not an int")
"not an int"
```

Here, *x* is not typ-checked, because it is excluded:

```
@typed(exclude=("x",))
def do_not_check_x(x: int, y: int, z: int) -> str:
    return ", ".join([str(x), str(y), str(z)])
```

```
>>> do_not_check_x("not an int", 2, 3)
"not an int, 2, 3"
```

The following function is effectively not type-checked, because the included parameter *x* is also excluded. (*exclude* prevails *include*):

```
@typed(exclude=("x", "y", "return"), include=("x",))
def effectively_check_nothing(x: int, y: float) -> str:
    return (x, y)
```

Example

Use *@typed* on a class: Instance methods and staticmethods are typed, even if they are inherited from an un-typed class; classmethods and nested classes are not typed.

```
class SomeSuperClass:
    def some_super_instance_method(self, x: int):
        pass

@typed
class SomeClass(SomeSuperClass):
    @classmethod
    def some_classmethod(cls, x: int):
        pass

    @staticmethod
    def some_staticmethod(cls, x: int):
        pass

    def __init__(self, x: int):
        pass

    def some_instance_method(self, x: int):
        pass

    class SomeNestedClass:
        def __init__(self, x: int):
            pass
```

```
>>> SomeClass("not an int")
RuntimeTypingError: TypingViolation in function `__init__`: Expected type of␣
↪argument `x` to be `<class 'int'>` (got `<class 'str'>`).
```

```
>>> SomeClass(1).some_instance_method("not an int")
RuntimeTypingError: TypingViolation in function `some_instance_method`: Expected␣
↪type of argument `x` to be `<class 'int'>` (got `<class 'str'>`)
```

```
>>> SomeClass(1).some_super_instance_method("not an int")
RuntimeTypingError: TypingViolation in function `some_super_instance_method`:␣
↪Expected type of argument `x` to be `<class 'int'>` (got `<class 'str'>`).
```

```
>>> SomeClass.some_staticmethod("not an int")
RuntimeTypingError: TypingViolation in function `some_staticmethod`: Expected type␣
↪of argument `x` to be `<class 'int'>` (got `<class 'str'>`).
```

```
>>> SomeClass.some_classmethod("not an int")  # does not raise
>>> SomeClass(1).SomeNestedClass("not an int")  # does not raise
>>> SomeClass.SomeNestedClass("not an int")  # does not raise
```

Example

Typing a classmethod. If you want to type a classmethod of a class, you can do so by explicitely decorating it:

```
class TypedClassMethodClass:
    @classmethod
    @typed
    def some_class_method(cls, x: int):
        pass
```

```
>>> TypedClassMethodClass.some_class_method("not an int")
RuntimeTypingError: TypingViolation in function `some_class_method`: Expected type␣
↪of argument `x` to be `<class 'int'>` (got `<class 'str'>`).
```

2.2 Violations

class runtime_typing.**RuntimeTypingViolation**(*obj: object*, *category: str*, *parameter_name: Any*, *expected: Any*, *got: Any*, *mode: Literal['raise', 'warn', 'return'] = 'raise'*, *defer: bool = False*)

Violation against Typing Annotation.

obj
: The object the violation occurred on.

parameter_name
: The name of the parameter the violation occurred on.

expected
: The expected value (or type) of the parameter.

got
: The actual value (or type) of the parameter.

message
: A human-readable message describing the violation. This is used in RuntimeTypingViolation.handle() when raising or warning.

class runtime_typing.**ComplexRuntimeTypingViolation**(*violations: List[runtime_typing.violations.RuntimeTypingViolation]*, *mode: Literal['raise', 'warn', 'return'] = 'raise'*, *defer: bool = False*, *conjunction: Literal['and', 'or'] = 'or'*)

Container of multiple TypingViolations.

violations
: List of the TypingViolations.

conjunction
: Whether the TypingViolations are AND- or OR-combined.

message
: A human-readable message used for raising and warning.

2.3 Exceptions / Warnings

exception runtime_typing.**RuntimeTypingError**

exception runtime_typing.**RuntimeTypingWarning**

PYTHON MODULE INDEX

INDEX