Python Type Annotations

How I Learned to Stop Worrying and Love Type Annotation

Gerard Keating
vfxger.com
PyCon Ireland 2023

Brief History

- 1990
  Guido invented a *dynamically* typed language called Python
- 2014
  Guido wrote PEP 484 – Type Hints and type annotations were introduced in Python ~3.6
- Today Python is still a *dynamically* typed language

```python
var1: int = "hello"

type(var1)
# <class 'str'>
```

```
class Foo:
  var1:int

try:
  Foo.var1
except AttributeError as err:
  print(err)
# AttributeError: type object 'Foo' has no attribute 'var1'
```

```
>>> class Foo:
...     var1:int
>>> Foo.__dict__
>>> mappingproxy(
            {'__module__': '__main__',
            '__annotations__': {'var1': int},
            '__dict__': <attribute '__dict__' of 'Foo' objects>,
            '__weakref__': <attribute '__weakref__' of 'Foo' objects>,
            '__doc__': None})
```

```
Foo.__annotations__
{'var1': int}
```

- Type annotations does not really affect the runnning of the code
- Useful as a form of structured documentation

> Incorrect documentation is often worse than no documentation

Bertrand Meyer

Use a Type Checker

```
var1: int = "hello"
```

`mypy exception_mypy.py`

exception_mypy.py:1: error: Incompatible types in assignment (expression has type "str", variable has type "int") [assignment]

Type Checkers

Use mypy

- Is the most mature and widely used
- It catches the most errors
- Alternatives:
  - Pytype -> Google
  - Pyright / Pylance -> Microsoft, vscode
  - Pyre -> Meta
  - Ruff -> Linter with some type checking
  - Others in your IDE

Other uses for Type Annotation

- dataclass (and attrs): use it to to help define classes
- pydantic: used for data validation
- fastAPI: uses pydantic and uses annotations for documentation
- SQLModel and other ORMs

Note on terminology

These refer to pretty much the same thing:

- Type Annotation
- Type Hinting
- Type Decorations
- Type Comments

For this presentation I'll be using the term **Type Annotation**

Note: Using Python 3.11

```
def new_way(bar:list[int])->int:
        return bar[5]
```

python 3.6 error:

```
Traceback (most recent call last):
  File "not_old_way.py", line 1, in <module>
    def new_way(bar: list[int]) -> int:
TypeError: 'type' object is not subscriptable
```

Note: Some great features in Python 3.12

Type Annotation Types

```
int_: int = 1
float_: float = 1.5
string_: str = "hello"
bytes_: bytes = b"bytes"
none: None = None

class Foo:
    ...
my_foo: Foo = Foo()
```

Inheritance

```
class Animal:
    ...

class Dog(Animal):
    ...

lassie: Dog = Dog()
bluey: Animal = Dog()

garfield: Dog = Animal()
# error: Incompatible types in assignment
# (expression has type "Animal", variable has type "Dog")  [assignment]
```

Inheritance `is a` relationship.

A dog `is a` animal

An animal is not necessarily a dog

Type Annotation Syntax: Union |

```
var1: int|float|str = 6
```

- `var1` is of type int or float or string
- Used to use `typing.Union`

Your own types

```
my_num = int|float
num: my_num = 6
```

Your type should have type annotation too

```
from typing import TypeAlias
my_text: TypeAlias = str|bytes
hello: my_text="Greetings"
```

Type Annotation Syntax: list

```
list1:list = [1]
list2:list[int] = [1]
list3:list[int|str] = [1, 'one']
list4:list[int|list[int]] = [1, [1]]
```

- `list1` is of type list
- `list2` is a list of ints
- `list3` is a list of ints or strings
- `list4` is a list of ints or lists of ints

Any

- `Any` means any type
- Generally usage is a code smell

```
from typing import Any
list1: list
list2: list[Any]
```

- `list1: list` is the same as `list2: list[Any]`

Type Annotation: dicts

```
dict1:dict = {"one":1}
# "dict" expects 2 type arguments (or no args)
dict2:dict[str, Any] = {"one":1}
dict3:dict[Any, int] = {"one":1}
dict4:dict[str, int|list[int|list[int]]] = {"one":1}
```

- `dict1` is the same as dict[Any, Any], any key type and any value type
- `dict2` is a dictionary with string type key and any value type
- `dict3` is a dictionary with string type key and any value type
- `dict4` you can work out yourself

Type Annotation: tuples

```
from typing import Any
simple: tuple = (1, 1,)
same_simple: tuple[Any, ...] = (1, 1, 's')
type_2int: tuple[int, int] = (1, 1,)
mix_types: tuple[int, str, float] = (1, 's', 0.5,)
all_ints: tuple[int, ...] = (1, 1, 1)
ints_strs: tuple[int | str, ...] = (1, 1, 's')
```

- tuple accepts any number of type arguments
- `...` as second argument denotes any length

Type Annotation Syntax: Function

```
def bar(foo: list[int]) -> int:
    return bar[0]
```

- bar takes a list of ints and returns an int

Type Annotation Syntax: classes

```
class Foo:
    bar: int # notice no value set
    val1: int | None = 7
    def __init__(self) -> None:
        ...
    def foo(self, arg1: int, arg2: str | int = 1) -> dict[str, int]:
        return {"one": 1}
```

- Tip: if you want the type checker to check `__init__` set its return type (to None)
- `self` does not need to have its type annotated

Typing Literal

```
from typing import Literal

def open_helper(mode: Literal["r", "rb", "w", "wb"]) -> Literal[True]:
    ...

open_helper("x") # ❌
```

mypy will now check the argument matches:

```
error: Argument 1 to "open_helper" has incompatible type "Literal['x']"; expected "Literal['r', 'rb', 'w', 'wb']"  [arg-type]

Found 1 error in 1 file (checked 1 source file)
```

Note: this is not a runtime check

The Problem

```
def say_hello_to_joe(email_sender) -> None:
    email_sender.send_email(
        to=['jo@example.com'],
        subject="Hello",
        text_body="Hi"
        )
```

Solution -> Protocol

```
from typing import Protocol


class EmailSender(Protocol):
    def send_email(self,
            to: list[str],
            subject: str,
            text_body: str) -> None:
        ...


def say_hello_to_joe( email_sender: EmailSender) -> None:
    email_sender.send_email(
        to=[email_address], subject="Hello", text_body="Hi")


class SendEmailRight:
    def send_email(self,
                to: list[str],
                subject: str,
                text_body: str) -> None:
        # actual implentation of sending email

say_hello_to_joe(SendEmailRight())
```

The Problem

```
def first_item_bad(items: list[Any]) -> Any:
    return items[0]


first_item_bad(["hello"]) + 1
```

- mypy will not complain
- But there is a TypeError

TypeVar

```
T = TypeVar('T')

def first_item_better(items: list[T]) -> T:
    return items[0]

first_item_better(["hello"]) + 1
# ^ error: Unsupported operand types for + ("str" and "int")  [operator]
first_item_better([1]) + 1 # no error
```

- Now mypy errors correctly

TypeVar adding constraints

```python
TItem = TypeVar('TItem', str, int)

def first_item_best(items: list[TItem]) -> TItem:
    return items[0]

first_item_best([None])
# ^ error: Value of type variable "TItem" of "first_item_best" cannot be "None"  [type-var]
first_item_best(["hello"])+1
# ^ error: Unsupported operand types for + ("str" and "int")  [operator]
first_item_best([1])+1
```

- mypy correctly errors on `first_item_best([None])` and `first_item_best(["hello"])+1`

typing.Final

- `Final` names cannot be reassigned in any scope

```
from typing import Final

DEBUG : Final[bool] = True

DEBUG = True # ❌ type check error
```

typing docs for more

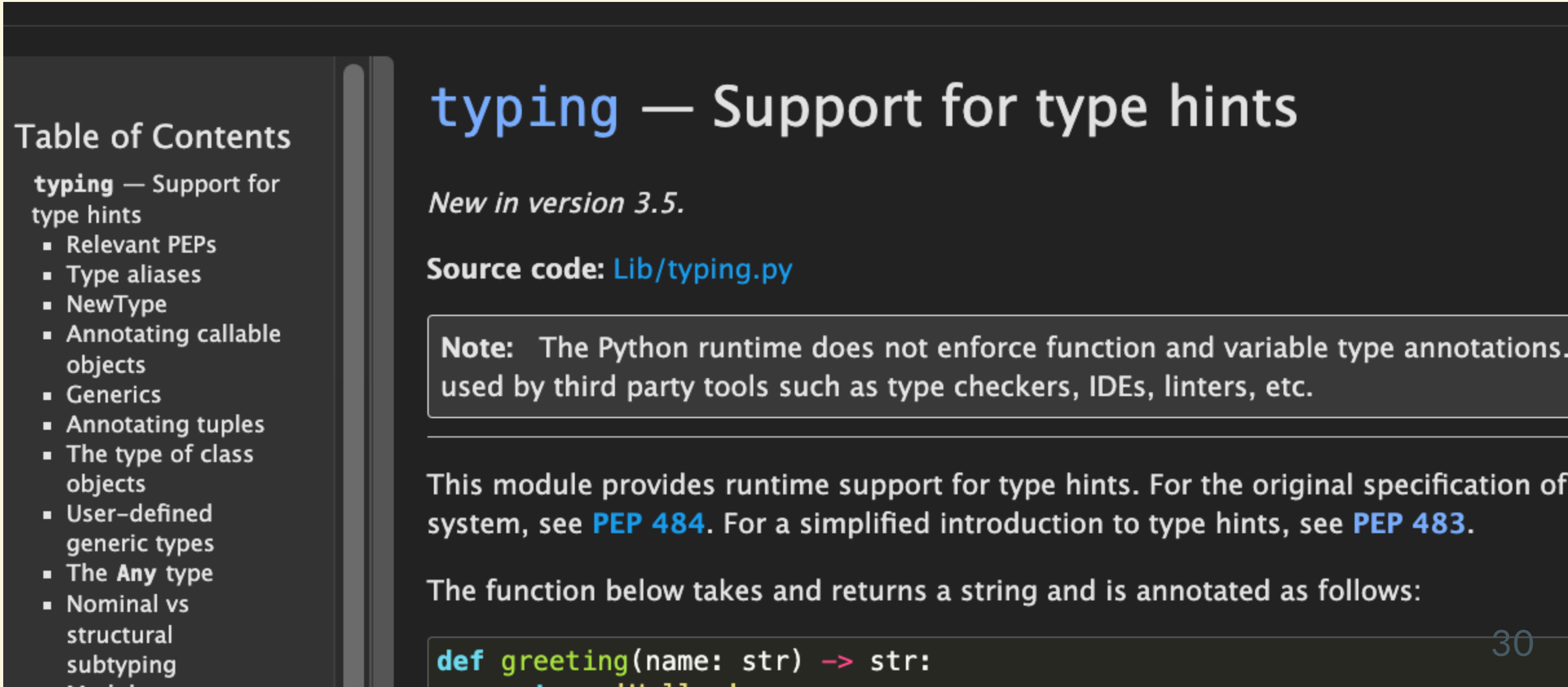


Table of Contents

typing — Support for type hints

New in version 3.5.

Source code: Lib/typing.py

Note: The Python runtime does not enforce function and variable type annotations. used by third party tools such as type checkers, IDEs, linters, etc.

This module provides runtime support for type hints. For the original specification of system, see **PEP 484**. For a simplified introduction to type hints, see **PEP 483**.

The function below takes and returns a string and is annotated as follows:

```
def greeting(name: str) -> str:
```

Stubs

- For packages you cannot alter the code of you can create stubs
- stubs already exist for many packages that are not typed

mypy will actually find some stubs for you

```
tokens.py:1: error: Library stubs not installed for "six" [import]
tokens.py:1: note: Hint: "python3 -m pip install types-six"
```

Example stub from stubs/Flask-Cors/flask_cors/core.pyi

```
def probably_regex(maybe_regex: str | Pattern[str]) -> bool: ...
def re_fix(reg: str) -> str: ...
def try_match_any(inst: str, patterns: Iterable[str | Pattern[str]]) -> bool: ...
```

Type Checking

- Type Annotation is **just** documentation
- Example, this code runs fine:

```
def foo()->dict[str:int]:
  ...
```

But the type annotation is wrong, check in mypy:

```
error: "dict" expects 2 type arguments, but 1 given  [type-arg]
error: Invalid type comment or annotation  [valid-type]
```

Running mypy

- Install mypy using pip or poetry or whatever
- Can be run on whole packages or single files and can be configured with exclude logic

```
mypy src
```

Misconceptions about mypy

- mypy will **not** complain about missing type annotations unless configured to do so
- mypy will **not** check the bodies of untyped functions unless configured to do so

```
def foo(bar=1):
  foobar: str = 'foobar'
  return bar + foobar
# note: By default the bodies of untyped
# functions are not checked,
# consider using --check-untyped-defs  [annotation-unchecked]
```

Getting started with type checking your code

1. Install mypy
2. Run mypy on your code
3. Fix mypy errors
4. Setup mypy configuration
5. Add mypy to your CI so it fails builds if any mypy errors happen
6. [Optionally] Sporadically manually run mypy strict rules and do some fixes

mypy config

> By default it uses the file mypy.ini with a fallback to .mypy.ini, then pyproject.toml, then setup.cfg

mypy.ini file

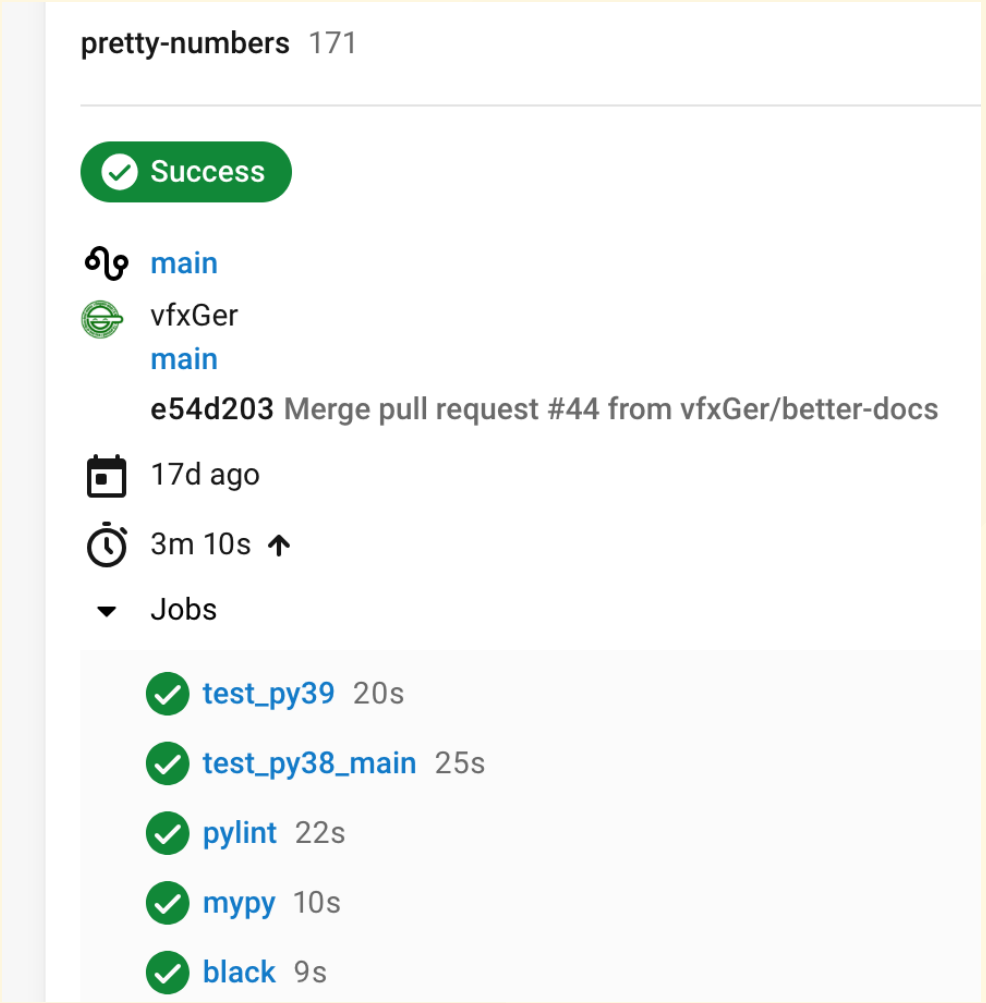
```
[mypy]
exclude = migrations/
ignore_missing_imports = True
```

CI/CD example

https://github.com/vfxGer/pretty-numbers

```
mypy:
  executor: python/default
  steps:
    - checkout
    - python/install-packages:
        pkg-manager: poetry
    - run:
        command: |
          poetry run mypy --strict pretty_numbers
```

CI/CD example



mypy --strict

- Misconception that --strict just checks if everything has a type annotation
- It actually enables all optional error checking flags which can change over time
- example flags:

```
--warn-unused-configs, --disallow-any-generics,
--disallow-subclassing-any --disallow-untyped-calls,
--disallow-untyped-defs, --disallow-
incomplete-defs,
--check-untyped-defs, --disallow-untyped-decorators, --warn-redundant-
casts, --warn-unused-ignores,
--warn-return-any, --no-implicit-reexport,
```

Why Type Annotation

- Documentation of code that helps other developers making your code more readable
- Enables IDE features that enables you to write code faster

Type checker finds and prevents BUGS

Questions?

Contact me vfxger.com