Mypy Documentation

Release 0.610+dev-0447473de2c00b5429ed0072249ecb4f134e2f38-dirty

Jukka

May 28, 2018

First steps

Mypy is a static type checker for Python 3 and Python 2.7.

CHAPTER 1

Introduction

Mypy is a static type checker for Python 3 and Python 2.7. If you sprinkle your code with type annotations, mypy can type check your code and find common bugs. As mypy is a static analyzer, or a lint-like tool, the type annotations are just hints for mypy and don't interfere when running your program. You run your program with a standard Python interpreter, and the annotations are treated effectively as comments.

Using the Python 3 function annotation syntax (using the PEP 484 notation) or a comment-based annotation syntax for Python 2 code, you will be able to efficiently annotate your code and use mypy to check the code for common errors. Mypy has a powerful and easy-to-use type system with modern features such as type inference, generics, function types, tuple types, and union types.

As a developer, you decide how to use mypy in your workflow. You can always escape to dynamic typing as mypy's approach to static typing doesn't restrict what you can do in your programs. Using mypy will make your programs easier to understand, debug, and maintain.

This documentation provides a short introduction to mypy. It will help you get started writing statically typed code. Knowledge of Python and a statically typed object-oriented language, such as Java, are assumed.

Note: Mypy is used in production by many companies and projects, but mypy is officially beta software. There will be occasional changes that break backward compatibility. The mypy development team tries to minimize the impact of changes to user code.

CHAPTER 2

Basics

This chapter introduces some core concepts of mypy, including function annotations, the `typing` module and library stubs. Read it carefully, as the rest of documentation may not make much sense otherwise.

2.1 Function signatures

A function without a type annotation is considered dynamically typed:

```
def greeting(name):
    return 'Hello, {}'.format(name)
```

You can declare the signature of a function using the Python 3 annotation syntax (Python 2 is discussed later in *Type checking Python 2 code*). This makes the function statically typed, and that causes type checker report type errors within the function.

Here's a version of the above function that is statically typed and will be type checked:

```
def greeting(name: str) -> str:
    return 'Hello, {}'.format(name)
```

If a function does not explicitly return a value we give the return type as `None`. Using a `None` result in a statically typed context results in a type check error:

```
def p() -> None:
    print('hello')

a = p()    # Type check error: p has None return value
```

Arguments with default values can be annotated as follows:

```
def greeting(name: str, prefix: str = 'Mr.') -> str:
   return 'Hello, {} {}'.format(name, prefix)
```

2.2 Running mypy

You can type check a program by using the `mypy` tool, which is basically a linter – it checks your program for errors without actually running it:

```
$ mypy program.py
```

All errors reported by mypy are essentially warnings that you are free to ignore, if you so wish.

The next chapter explains how to download and install mypy: *Getting started*.

More command line options are documented in *The mypy command line*.

Note: Depending on how mypy is configured, you may have to run mypy like this:

```
$ python3 -m mypy program.py
```

2.3 The typing module

The `typing` module contains many definitions that are useful in statically typed code. You typically use `from ... import` to import them (we'll explain `Iterable` later in this document):

```
from typing import Iterable

def greet_all(names: Iterable[str]) -> None:
    for name in names:
        print('Hello, {}'.format(name))
```

For brevity, we often omit the `typing` import in code examples, but mypy will give an error if you use definitions such as `Iterable` without first importing them.

2.4 Mixing dynamic and static typing

Mixing dynamic and static typing within a single file is often useful. For example, if you are migrating existing Python code to static typing, it may be easiest to do this incrementally, such as by migrating a few functions at a time. Also, when prototyping a new feature, you may decide to first implement the relevant code using dynamic typing and only add type signatures later, when the code is more stable.

```
def f():
    1 + 'x'  # No static type error (dynamically typed)

def g() -> None:
    1 + 'x'  # Type check error (statically typed)
```

Note: The earlier stages of mypy, known as the semantic analysis, may report errors even for dynamically typed functions. However, you should not rely on this, as this may change in the future.

2.5 Library stubs and the Typeshed repo

In order to type check code that uses library modules such as those included in the Python standard library, you need to have library *stubs*. A library stub defines a skeleton of the public interface of the library, including classes, variables and functions and their types, but dummy function bodies.

For example, consider this code:

```
x = chr(4)
```

Without a library stub, the type checker would have no way of inferring the type of `x` and checking that the argument to `chr` has a valid type. Mypy incorporates the typeshed project, which contains library stubs for the Python builtins and the standard library. The stub for the builtins contains a definition like this for `chr`:

```
def chr(code: int) -> str: ...
```

In stub files we don't care about the function bodies, so we use an ellipsis instead. That `...` is three literal dots!

Mypy complains if it can't find a stub (or a real module) for a library module that you import. You can create a stub easily; here is an overview:

- Write a stub file for the library and store it as a `.pyi` file in the same directory as the library module.
- Alternatively, put your stubs (`.pyi` files) in a directory reserved for stubs (e.g., `myproject/stubs`). In this case you have to set the environment variable `MYPYPATH` to refer to the directory. For example:

  ```
  $ export MYPYPATH=~/work/myproject/stubs
  ```

Use the normal Python file name conventions for modules, e.g. `csv.pyi` for module `csv`. Use a subdirectory with `__init__.pyi` for packages.

If a directory contains both a `.py` and a `.pyi` file for the same module, the `.pyi` file takes precedence. This way you can easily add annotations for a module even if you don't want to modify the source code. This can be useful, for example, if you use 3rd party open source libraries in your program (and there are no stubs in typeshed yet).

That's it! Now you can access the module in mypy programs and type check code that uses the library. If you write a stub for a library module, consider making it available for other programmers that use mypy by contributing it back to the typeshed repo.

There is more information about creating stubs in the mypy wiki. The following sections explain the kinds of type annotations you can use in your programs and stub files.

Note: You may be tempted to point `MYPYPATH` to the standard library or to the `site-packages` directory where your 3rd party packages are installed. This is almost always a bad idea – you will likely get tons of error messages about code you didn't write and that mypy can't analyze all that well yet, and in the worst case scenario mypy may crash due to some construct in a 3rd party package that it didn't expect.

CHAPTER 3

Getting started

3.1 Installation

Mypy requires Python 3.4 or later to run. Once you've installed Python 3, you can install mypy with:

```
$ python3 -m pip install mypy
```

Note that even though you need Python 3 to run `mypy`, type checking Python 2 code is fully supported, as discussed in *Type checking Python 2 code*.

3.2 Installing from source

To install mypy from source, clone the mypy repository on GitHub and then run `pip install` locally:

```
$ git clone --recurse-submodules https://github.com/python/mypy.git
$ cd mypy
$ sudo python3 -m pip install --upgrade .
```

CHAPTER 4

Type hints cheat sheet (Python 3)

This document is a quick cheat sheet showing how the PEP 484 type language represents various common types in Python 3. Unless otherwise noted, the syntax is valid on all versions of Python 3.

Note: Technically many of the type annotations shown below are redundant, because mypy can derive them from the type of the expression. So many of the examples have a dual purpose: show how to write the annotation, and show the inferred types.

4.1 Built-in types

```
from typing import List, Set, Dict, Tuple, Text, Optional, AnyStr

# For simple built-in types, just use the name of the type.
x = 1  # type: int
x = 1.0  # type: float
x = True  # type: bool
x = "test"  # type: str
x = u"test"  # type: str
x = b"test"  # type: bytes

# For collections, the name of the type is capitalized, and the
# name of the type inside the collection is in brackets.
x = [1]  # type: List[int]
x = {6, 7}  # type: Set[int]

# Empty Tuple types are a bit special
x = ()  # type: Tuple[()]

# For mappings, we need the types of both keys and values.
x = {'field': 2.0}  # type: Dict[str, float]
```

(continues on next page)

(continued from previous page)

```
# For tuples, we specify the types of all the elements.
x = (3, "yes", 7.5)  # type: Tuple[int, str, float]

# For textual data, use Text.
# This is `unicode` in Python 2 and `str` in Python 3.
x = ["string", u"unicode"]  # type: List[Text]



# Use Optional for values that could be None.
input_str = f()  # type: Optional[str]
if input_str is not None:
   print(input_str)
```

4.2 Functions

Python 3 introduces an annotation syntax for function declarations in PEP 3107.

```
from typing import Callable, Iterable, Union, Optional, List

# This is how you annotate a function definition.
def stringify(num: int) -> str:
    return str(num)

# And here's how you specify multiple arguments.
def plus(num1: int, num2: int) -> int:
    return num1 + num2

# Add type annotations for kwargs as though they were positional args.
def f(num1: int, my_float: float = 3.5) -> float:
    return num1 + my_float

# An argument can be declared positional-only by giving it a name
# starting with two underscores:
def quux(__x: int) -> None:
    pass
quux(3)  # Fine
quux(__x=3)  # Error

# This is how you annotate a function value.
x = f # type: Callable[[int, float], float]

# A generator function that yields ints is secretly just a function that
# returns an iterable (see below) of ints, so that's how we annotate it.
def f(n: int) -> Iterable[int]:
    i = 0
    while i < n:
        yield i
        i += 1

# For a function with many arguments, you can of course split it over multiple lines
def send_email(address: Union[str, List[str]],
               sender: str,
               cc: Optional[List[str]],
```

(continues on next page)

(continued from previous page)

```
                bcc: Optional[List[str]],
                subject='',
                body: List[str] = None
                ) -> bool:

    ...
```

4.3 Coroutines and asyncio

See *Typing async/await* for the full detail on typing coroutines and asynchronous code.

```
import asyncio
from typing import Generator, Any

# A generator-based coroutine created with @asyncio.coroutine should have a
# return type of Generator[Any, None, T], where T is the type it returns.
@asyncio.coroutine
def countdown34(tag: str, count: int) -> Generator[Any, None, str]:
    while count > 0:
        print('T-minus {} ({})'.format(count, tag))
        yield from asyncio.sleep(0.1)
        count -= 1
    return "Blastoff!"

# mypy currently does not support converting functions into generator-based
# coroutines in Python 3.4, so you need to add a 'yield' to make it
# typecheck.
@asyncio.coroutine
def async1(obj: object) -> Generator[None, None, str]:
    if False:
        yield
    return "placeholder"

# A Python 3.5+ coroutine is typed like a normal function.
async def countdown35(tag: str, count: int) -> str:
    while count > 0:
        print('T-minus {} ({})'.format(count, tag))
        await asyncio.sleep(0.1)
        count -= 1
    return "Blastoff!"

async def async2(obj: object) -> str:
    return "placeholder"
```

4.4 When you're puzzled or when things are complicated

```
from typing import Union, Any, List, cast

# To find out what type mypy infers for an expression anywhere in
# your program, wrap it in reveal_type.  Mypy will print an error
# message with the type; remove it again before running the code.
```

(continues on next page)

(continued from previous page)

```
reveal_type(1)  # -> error: Revealed type is 'builtins.int'

# Use Union when something could be one of a few types.
x = [3, 5, "test", "fun"]  # type: List[Union[int, str]]

# Use Any if you don't know the type of something or it's too
# dynamic to write a type for.
x = mystery_function()  # type: Any

# This is how to deal with varargs.
# This makes each positional arg and each keyword arg a 'str'.
def call(self, *args: str, **kwargs: str) -> str:
        request = make_request(*args, **kwargs)
        return self.do_api_query(request)

# Use `ignore` to suppress type-checking on a given line, when your
# code confuses mypy or runs into an outright bug in mypy.
# Good practice is to comment every `ignore` with a bug link
# (in mypy, typeshed, or your own code) or an explanation of the issue.
x = confusing_function()  # type: ignore # https://github.com/python/mypy/issues/1167

# cast is a helper function for mypy that allows for guidance of how to convert types.
# it does not cast at runtime
a = [4]
b = cast(List[int], a)  # passes fine
c = cast(List[str], a)  # passes fine (no runtime check)
reveal_type(c)  # -> error: Revealed type is 'builtins.list[builtins.str]'
print(c)  # -> [4] the object is not cast

# if you want dynamic attributes on your class, have it override __setattr__ or __
↪getattr__
# in a stub or in your source code.
# __setattr__ allows for dynamic assignment to names
# __getattr__ allows for dynamic access to names
class A:
    # this will allow assignment to any A.x, if x is the same type as `value`
    def __setattr__(self, name: str, value: int) -> None: ...
    # this will allow access to any A.x, if x is compatible with the return type
    def __getattr__(self, name: str) -> int: ...
a.foo = 42  # works
a.bar = 'Ex-parrot'  # fails type checking


# TODO: explain "Need type annotation for variable" when
# initializing with None or an empty container
```

4.5 Standard duck types

In typical Python code, many functions that can take a list or a dict as an argument only need their argument to be somehow "list-like" or "dict-like". A specific meaning of "list-like" or "dict-like" (or something-else-like) is called a "duck type", and several duck types that are common in idiomatic Python are standardized.

```
from typing import Mapping, MutableMapping, Sequence, Iterable, List, Set
```

(continues on next page)

(continued from previous page)

```
# Use Iterable for generic iterables (anything usable in `for`),
# and Sequence where a sequence (supporting `len` and `__getitem__`) is required.
def f(iterable_of_ints: Iterable[int]) -> List[str]:
    return [str(x) for x in iterable_of_ints]
f(range(1, 3))

# Mapping describes a dict-like object (with `__getitem__`) that we won't mutate,
# and MutableMapping one (with `__setitem__`) that we might.
def f(my_dict: Mapping[int, str])-> List[int]:
    return list(my_dict.keys())
f({3: 'yes', 4: 'no'})
def f(my_mapping: MutableMapping[int, str]) -> Set[str]:
    my_mapping[5] = 'maybe'
    return set(my_mapping.values())
f({3: 'yes', 4: 'no'})
```

4.6 Classes

```
class MyClass:
    # The __init__ method doesn't return anything, so it gets return
    # type None just like any other method that doesn't return anything.
    def __init__(self) -> None:
        ...
    # For instance methods, omit `self`.
    def my_method(self, num: int, str1: str) -> str:
        return num * str1



# User-defined classes are written with just their own names.
x = MyClass() # type: MyClass
```

4.7 Other stuff

```
import sys
import re
# typing.Match describes regex matches from the re module.
from typing import Match, AnyStr, IO
x = re.match(r'[0-9]+', "15")  # type: Match[str]

# You can use AnyStr to indicate that any string type will work
# but not to mix types
def full_name(first: AnyStr, last: AnyStr) -> AnyStr:
    return first+last
full_name('Jon','Doe')  # same str ok
full_name(b'Bill', b'Bit')  # same binary ok
full_name(b'Terry', 'Trouble')  # different str types, fails

# Use IO[] for functions that should accept or return any
# object that comes from an open() call. The IO[] does not
# distinguish between reading, writing or other modes.
```

(continues on next page)

(continued from previous page)

```
def get_sys_IO(mode='w') -> IO[str]:
    if mode == 'w':
        return sys.stdout
    elif mode == 'r':
        return sys.stdin
    else:
        return sys.stdout

# forward references are useful if you want to reference a class before it is designed

def f(foo: A) -> int:  # this will fail
    ...

class A:
    ...

# however, using the string 'A', it will pass as long as there is a class of that␣
↪name later on
def f(foo: 'A') -> int:
    ...

# TODO: add TypeVar and a simple generic function
```

4.8 Variable Annotation in Python 3.6 with PEP 526

Python 3.6 brings new syntax for annotating variables with PEP 526. Mypy brings limited support for PEP 526 annotations.

```
# annotation is similar to arguments to functions
name: str = "Eric Idle"

# class instances can be annotated as follows
mc : MyClass = MyClass()

# tuple packing can be done as follows
tu: Tuple[str, ...] = ('a', 'b', 'c')

# annotations are not checked at runtime
year: int = '1972'  # error in type checking, but works at runtime

# these are all equivalent
hour = 24 # type: int
hour: int; hour = 24
hour: int = 24

# you do not (!) need to initialize a variable to annotate it
a: int # ok for type checking and runtime

# which is useful in conditional branches
child: bool
if age < 18:
    child = True
else:
    child = False
```

(continues on next page)

(continued from previous page)

```
# annotations for classes are for instance variables (those created in __init__ or __
↪new__)
class Battery:
    charge_percent: int = 100  # this is an instance variable with a default value
    capacity: int  # an instance variable without a default

# you can use the ClassVar annotation to make the variable a class variable instead␣
↪of an instance variable.
class Car:
    seats: ClassVar[int] = 4
    passengers: ClassVar[List[str]]

 # You can also declare the type of an attribute in __init__
 class Box:
     def __init__(self) -> None:
         self.items: List[str] = []
```

Please see *New features in Python 3.6* for more on mypy's compatibility with Python 3.6's new features.

CHAPTER 5

Type hints cheat sheet (Python 2)

This document is a quick cheat sheet showing how the PEP 484 type language represents various common types in Python 2.

Note: Technically many of the type annotations shown below are redundant, because mypy can derive them from the type of the expression. So many of the examples have a dual purpose: show how to write the annotation, and show the inferred types.

5.1 Built-in types

```
from typing import List, Set, Dict, Tuple, Text, Optional

# For simple built-in types, just use the name of the type.
x = 1 # type: int
x = 1.0 # type: float
x = True # type: bool
x = "test" # type: str
x = u"test" # type: unicode

# For collections, the name of the type is capitalized, and the
# name of the type inside the collection is in brackets.
x = [1] # type: List[int]
x = set([6, 7]) # type: Set[int]

# Empty Tuple types are a bit special
x = ()  # type: Tuple[()]

# For mappings, we need the types of both keys and values.
x = dict(field=2.0) # type: Dict[str, float]

# For tuples, we specify the types of all the elements.
```

(continues on next page)

(continued from previous page)

```
x = (3, "yes", 7.5) # type: Tuple[int, str, float]

# For textual data, use Text.
# This is `unicode` in Python 2 and `str` in Python 3.
x = ["string", u"unicode"] # type: List[Text]

# Use Optional for values that could be None.
input_str = f() # type: Optional[str]
if input_str is not None:
   print input_str
```

5.2 Functions

```
from typing import Callable, Iterable

# This is how you annotate a function definition.
def stringify(num):
    # type: (int) -> str
    """Your function docstring goes here after the type definition."""
    return str(num)

# This function has no parameters and also returns nothing. Annotations
# can also be placed on the same line as their function headers.
def greet_world(): # type: () -> None
    print "Hello, world!"

# And here's how you specify multiple arguments.
def plus(num1, num2):
    # type: (int, int) -> int
    return num1 + num2

# Add type annotations for kwargs as though they were positional args.
def f(num1, my_float=3.5):
    # type: (int, float) -> float
    return num1 + my_float

# An argument can be declared positional-only by giving it a name
# starting with two underscores:
def quux(__x):
    # type: (int) -> None
    pass
quux(3)  # Fine
quux(__x=3)  # Error

# This is how you annotate a function value.
x = f # type: Callable[[int, float], float]

# A generator function that yields ints is secretly just a function that
# returns an iterable (see below) of ints, so that's how we annotate it.
def f(n):
    # type: (int) -> Iterable[int]
    i = 0
    while i < n:
        yield i
```

(continues on next page)

(continued from previous page)

```
        i += 1

# There's alternative syntax for functions with many arguments.
def send_email(address,     # type: Union[str, List[str]]
               sender,      # type: str
               cc,          # type: Optional[List[str]]
               bcc,         # type: Optional[List[str]]
               subject='',
               body=None    # type: List[str]
               ):
    # type: (...) -> bool
     <code>
```

5.3 When you're puzzled or when things are complicated

```
from typing import Union, Any, cast

# To find out what type mypy infers for an expression anywhere in
# your program, wrap it in reveal_type.  Mypy will print an error
# message with the type; remove it again before running the code.
reveal_type(1) # -> error: Revealed type is 'builtins.int'

# Use Union when something could be one of a few types.
x = [3, 5, "test", "fun"] # type: List[Union[int, str]]

# Use Any if you don't know the type of something or it's too
# dynamic to write a type for.
x = mystery_function() # type: Any

# This is how to deal with varargs.
# This makes each positional arg and each keyword arg a 'str'.
def call(self, *args, **kwargs):
         # type: (*str, **str) -> str
         request = make_request(*args, **kwargs)
         return self.do_api_query(request)

# Use `ignore` to suppress type-checking on a given line, when your
# code confuses mypy or runs into an outright bug in mypy.
# Good practice is to comment every `ignore` with a bug link
# (in mypy, typeshed, or your own code) or an explanation of the issue.
x = confusing_function() # type: ignore # https://github.com/python/mypy/issues/1167

# cast is a helper function for mypy that allows for guidance of how to convert types.
# it does not cast at runtime
a = [4]
b = cast(List[int], a)  # passes fine
c = cast(List[str], a)  # passes fine (no runtime check)
reveal_type(c)  # -> error: Revealed type is 'builtins.list[builtins.str]'
print(c)  # -> [4] the object is not cast

# if you want dynamic attributes on your class, have it override __setattr__ or __
↪getattr__
# in a stub or in your source code.
# __setattr__ allows for dynamic assignment to names
```

(continues on next page)

(continued from previous page)

```
# __getattr__ allows for dynamic access to names
class A:
    # this will allow assignment to any A.x, if x is the same type as `value`
    def __setattr__(self, name, value):
        # type: (str, int) -> None
        ...
a.foo = 42  # works
a.bar = 'Ex-parrot'  # fails type checking

# TODO: explain "Need type annotation for variable" when
# initializing with None or an empty container
```

5.4 Standard duck types

In typical Python code, many functions that can take a list or a dict as an argument only need their argument to be somehow "list-like" or "dict-like". A specific meaning of "list-like" or "dict-like" (or something-else-like) is called a "duck type", and several duck types that are common in idiomatic Python are standardized.

```
from typing import Mapping, MutableMapping, Sequence, Iterable

# Use Iterable for generic iterables (anything usable in `for`),
# and Sequence where a sequence (supporting `len` and `__getitem__`) is required.
def f(iterable_of_ints):
    # type: (Iterable[int]) -> List[str]
    return [str(x) for x in iterator_of_ints]
f(range(1, 3))

# Mapping describes a dict-like object (with `__getitem__`) that we won't mutate,
# and MutableMapping one (with `__setitem__`) that we might.
def f(my_dict):
    # type: (Mapping[int, str]) -> List[int]
    return list(my_dict.keys())
f({3: 'yes', 4: 'no'})
def f(my_mapping):
    # type: (MutableMapping[int, str]) -> Set[str]
    my_dict[5] = 'maybe'
    return set(my_dict.values())
f({3: 'yes', 4: 'no'})
```

5.5 Classes

```
class MyClass(object):

    # For instance methods, omit `self`.
    def my_method(self, num, str1):
        # type: (int, str) -> str
        return num * str1

    # The __init__ method doesn't return anything, so it gets return
    # type None just like any other method that doesn't return anything.
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        # type: () -> None
        pass

# User-defined classes are written with just their own names.
x = MyClass() # type: MyClass
```

5.6 Other stuff

```
import sys
# typing.Match describes regex matches from the re module.
from typing import Match, AnyStr, IO
x = re.match(r'[0-9]+', "15") # type: Match[str]

# Use AnyStr for functions that should accept any kind of string
# without allowing different kinds of strings to mix.
def concat(a, b):
    # type: (AnyStr, AnyStr) -> AnyStr
    return a + b
concat(u"foo", u"bar")  # type: unicode
concat(b"foo", b"bar")  # type: bytes

# Use IO[] for functions that should accept or return any
# object that comes from an open() call. The IO[] does not
# distinguish between reading, writing or other modes.
def get_sys_IO(mode='w'):
    # type: (str) -> IO[str]
    if mode == 'w':
        return sys.stdout
    elif mode == 'r':
        return sys.stdin
    else:
        return sys.stdout

# TODO: add TypeVar and a simple generic function
```

CHAPTER 6

Type checking Python 2 code

For code that needs to be Python 2.7 compatible, function type annotations are given in comments, since the function annotation syntax was introduced in Python 3. The comment-based syntax is specified in PEP 484.

Run mypy in Python 2 mode by using the `--py2` option:

```
$ mypy --py2 program.py
```

To run your program, you must have the `typing` module in your Python 2 module search path. Use `pip install typing` to install the module. This also works for Python 3 versions prior to 3.5 that don't include `typing` in the standard library.

The example below illustrates the Python 2 function type annotation syntax. This syntax is also valid in Python 3 mode:

```
from typing import List

def hello(): # type: () -> None
    print 'hello'

class Example:
    def method(self, lst, opt=0, *args, **kwargs):
        # type: (List[str], int, *str, **bool) -> int
        """Docstring comes after type comment."""
        ...
```

It's worth going through these details carefully to avoid surprises:

- You don't provide an annotation for the `self` / `cls` variable of methods.
- Docstring always comes *after* the type comment.
- For `*args` and `**kwargs` the type should be prefixed with `*` or `**`, respectively (except when using the multi-line annotation syntax described below). Again, the above example illustrates this.
- Things like `Any` must be imported from `typing`, even if they are only used in comments.

- In Python 2 mode `str` is implicitly promoted to `unicode`, similar to how `int` is compatible with `float`. This is unlike `bytes` and `str` in Python 3, which are incompatible. `bytes` in Python 2 is equivalent to `str`. (This might change in the future.)

6.1 Multi-line Python 2 function annotations

Mypy also supports a multi-line comment annotation syntax. You can provide a separate annotation for each argument using the variable annotation syntax. When using the single-line annotation syntax described above, functions with long argument lists tend to result in overly long type comments and it's often tricky to see which argument type corresponds to which argument. The alternative, multi-line annotation syntax makes long annotations easier to read and write.

Here is an example (from PEP 484):

```
def send_email(address,      # type: Union[str, List[str]]
               sender,       # type: str
               cc,           # type: Optional[List[str]]
               bcc,          # type: Optional[List[str]]
               subject='',
               body=None     # type: List[str]
               ):
    # type: (...) -> bool
    """Send an email message.  Return True if successful."""
    <code>
```

You write a separate annotation for each function argument on the same line as the argument. Each annotation must be on a separate line. If you leave out an annotation for an argument, it defaults to `Any`. You provide a return type annotation in the body of the function using the form `# type: (...) -> rt`, where `rt` is the return type. Note that the return type annotation contains literal three dots.

When using multi-line comments, you do not need to prefix the types of your `*arg` and `**kwarg` parameters with `*` or `**`. For example, here is how you would annotate the first example using multi-line comments:

```
from typing import List

class Example:
    def method(self,
               lst,       # type: List[str]
               opt=0,     # type: int
               *args,     # type: str
               **kwargs   # type: bool
               ):
        # type: (...) -> int
        """Docstring comes after type comment."""
        ...
```

6.2 Additional notes

- You should include types for arguments with default values in the annotation. The `opt` argument of `method` in the example at the beginning of this section is an example of this.
- The annotation can be on the same line as the function header or on the following line.
- Variables use a comment-based type syntax (explained in *Explicit types for variables*).

- You don't need to use string literal escapes for forward references within comments (string literal escapes are explained later).
- Mypy uses a separate set of library stub files in typeshed for Python 2. Library support may vary between Python 2 and Python 3.

CHAPTER 7

Built-in types

These are examples of some of the most common built-in types:

Type	Description
`int`	integer
`float`	floating point number
`bool`	boolean value
`str`	string (unicode)
`bytes`	8-bit string
`object`	an arbitrary object (`object` is the common base class)
`List[str]`	list of `str` objects
`Tuple[int, int]`	tuple of two `int` objects (`Tuple[()]` is the empty tuple)
`Tuple[int, ...]`	tuple of an arbitrary number of `int` objects
`Dict[str, int]`	dictionary from `str` keys to `int` values
`Iterable[int]`	iterable object containing ints
`Sequence[bool]`	sequence of booleans (read-only)
`Mapping[str, int]`	mapping from `str` keys to `int` values (read-only)
`Any`	dynamically typed value with an arbitrary type

The type `Any` and type constructors such as `List`, `Dict`, `Iterable` and `Sequence` are defined in the `typing` module.

The type `Dict` is a *generic* class, signified by type arguments within `[...]`. For example, `Dict[int, str]` is a dictionary from integers to strings and and `Dict[Any, Any]` is a dictionary of dynamically typed (arbitrary) values and keys. `List` is another generic class. `Dict` and `List` are aliases for the built-ins `dict` and `list`, respectively.

`Iterable`, `Sequence`, and `Mapping` are generic types that correspond to Python protocols. For example, a `str` object or a `List[str]` object is valid when `Iterable[str]` or `Sequence[str]` is expected. Note that even though they are similar to abstract base classes defined in `abc.collections` (formerly `collections`), they are not identical, since the built-in collection type objects do not support indexing.

CHAPTER 8

Type inference and type annotations

8.1 Type inference

Mypy considers the initial assignment as the definition of a variable. If you do not explicitly specify the type of the variable, mypy infers the type based on the static type of the value expression:

```
i = 1           # Infer type "int" for i
l = [1, 2]      # Infer type "List[int]" for l
```

Type inference is not used in dynamically typed functions (those without a function type annotation) — every local variable type defaults to `Any` in such functions. `Any` is discussed later in more detail.

8.2 Explicit types for variables

You can override the inferred type of a variable by using a variable type annotation:

```
from typing import Union

x: Union[int, str] = 1
```

Without the type annotation, the type of `x` would be just `int`. We use an annotation to give it a more general type `Union[int, str]` (this type means that the value can be either an `int` or a `str`). Mypy checks that the type of the initializer is compatible with the declared type. The following example is not valid, since the initializer is a floating point number, and this is incompatible with the declared type:

```
x: Union[int, str] = 1.1  # Error!
```

The variable annotation syntax is available starting from Python 3.6. In earlier Python versions, you can use a special comment after an assignment statement to declare the type of a variable:

```
x = 1  # type: Union[int, str]
```

We'll use both syntax variants in examples. The syntax variants are mostly interchangeable, but the variable annotation syntax allows defining the type of a variable without initialization, which is not possible with the comment syntax:

```
x: str  # Declare type of 'x' without initialization
```

Note: The best way to think about this is that the type annotation sets the type of the variable, not the type of the expression. To force the type of an expression you can use `cast(<type>, <expression>)`.

8.3 Explicit types for collections

The type checker cannot always infer the type of a list or a dictionary. This often arises when creating an empty list or dictionary and assigning it to a new variable that doesn't have an explicit variable type. Here is an example where mypy can't infer the type without some help:

```
l = []  # Error: Need type annotation for 'l'
```

In these cases you can give the type explicitly using a type annotation:

```
l: List[int] = []       # Create empty list with type List[int]
d: Dict[str, int] = {}  # Create empty dictionary (str -> int)
```

Similarly, you can also give an explicit type when creating an empty set:

```
s: Set[int] = set()
```

8.4 Compatibility of container types

The following program generates a mypy error, since `List[int]` is not compatible with `List[object]`:

```
def f(l: List[object], k: List[int]) -> None:
    l = k  # Type check error: incompatible types in assignment
```

The reason why the above assignment is disallowed is that allowing the assignment could result in non-int values stored in a list of `int`:

```
def f(l: List[object], k: List[int]) -> None:
    l = k
    l.append('x')
    print(k[-1])  # Ouch; a string in List[int]
```

Other container types like `Dict` and `Set` behave similarly. We will discuss how you can work around this in *Invariance vs covariance*.

You can still run the above program; it prints `x`. This illustrates the fact that static types are used during type checking, but they do not affect the runtime behavior of programs. You can run programs with type check failures, which is often very handy when performing a large refactoring. Thus you can always 'work around' the type system, and it doesn't really limit what you can do in your program.

8.5 Context in type inference

Type inference is *bidirectional* and takes context into account. For example, the following is valid:

```
def f(l: List[object]) -> None:
    l = [1, 2]  # Infer type List[object] for [1, 2], not List[int]
```

In an assignment, the type context is determined by the assignment target. In this case this is `l`, which has the type `List[object]`. The value expression `[1, 2]` is type checked in this context and given the type `List[object]`. In the previous example we introduced a new variable `l`, and here the type context was empty.

Declared argument types are also used for type context. In this program mypy knows that the empty list `[]` should have type `List[int]` based on the declared type of `arg` in `foo`:

```
def foo(arg: List[int]) -> None:
    print('Items:', ''.join(str(a) for a in arg))

foo([])  # OK
```

However, context only works within a single statement. Here mypy requires an annotation for the empty list, since the context would only be available in the following statement:

```
def foo(arg: List[int]) -> None:
    print(Items: ', '.join(arg))

a = []  # Error: Need type annotation for 'a'
foo(a)
```

Working around the issue is easy by adding a type annotation:

```
...
a: List[int] = []  # OK
foo(a)
```

8.6 Declaring multiple variable types at a time

You can declare more than a single variable at a time, but only with a type comment. In order to nicely work with multiple assignment, you must give each variable a type separately:

```
i, found = 0, False # type: int, bool
```

You can optionally use parentheses around the types, assignment targets and assigned expression:

```
i, found = 0, False # type: (int, bool)       # OK
(i, found) = 0, False # type: int, bool       # OK
i, found = (0, False) # type: int, bool       # OK
(i, found) = (0, False) # type: (int, bool)   # OK
```

8.7 Starred expressions

In most cases, mypy can infer the type of starred expressions from the right-hand side of an assignment, but not always:

```
a, *bs = 1, 2, 3    # OK
p, q, *rs = 1, 2    # Error: Type of rs cannot be inferred
```

On first line, the type of `bs` is inferred to be `List[int]`. However, on the second line, mypy cannot infer the type of `rs`, because there is no right-hand side value for `rs` to infer the type from. In cases like these, the starred expression needs to be annotated with a starred type:

```
p, q, *rs = 1, 2  # type: int, int, *List[int]
```

Here, the type of `rs` is set to `List[int]`.

8.8 Types in stub files

Stub files are written in normal Python 3 syntax, but generally leaving out runtime logic like variable initializers, function bodies, and default arguments, or replacing them with ellipses.

In this example, each ellipsis `...` is literally written in the stub file as three dots:

```
x: int

def afunc(code: str) -> int: ...
def afunc(a: int, b: int = ...) -> int: ...
```

Note: The ellipsis `...` is also used with a different meaning in *callable types* and *tuple types*.

CHAPTER 9

Kinds of types

9.1 User-defined types

Each class is also a type. Any instance of a subclass is also compatible with all superclasses. All values are compatible with the `object` type (and also the `Any` type).

```
class A:
    def f(self) -> int:         # Type of self inferred (A)
        return 2

class B(A):
    def f(self) -> int:
         return 3
    def g(self) -> int:
        return 4

a = B() # type: A  # OK (explicit type for a; override type inference)
print(a.f())       # 3
a.g()              # Type check error: A has no method g
```

9.2 The Any type

A value with the `Any` type is dynamically typed. Mypy doesn't know anything about the possible runtime types of such value. Any operations are permitted on the value, and the operations are checked at runtime, similar to normal Python code without type annotations.

`Any` is compatible with every other type, and vice versa. No implicit type check is inserted when assigning a value of type `Any` to a variable with a more precise type:

```
a = None  # type: Any
s = ''    # type: str
```

(continues on next page)

(continued from previous page)

```
a = 2      # OK
s = a      # OK
```

Declared (and inferred) types are *erased* at runtime. They are basically treated as comments, and thus the above code does not generate a runtime error, even though `s` gets an `int` value when the program is run. Note that the declared type of `s` is actually `str`!

If you do not define a function return value or argument types, these default to `Any`:

```
def show_heading(s) -> None:
    print('=== ' + s + ' ===')  # No static type checking, as s has type Any

show_heading(1)  # OK (runtime error only; mypy won't generate an error)
```

You should give a statically typed function an explicit `None` return type even if it doesn't return a value, as this lets mypy catch additional type errors:

```
def wait(t: float):  # Implicit Any return value
    print('Waiting...')
    time.sleep(t)

if wait(2) > 1:   # Mypy doesn't catch this error!
    ...
```

If we had used an explicit `None` return type, mypy would have caught the error:

```
def wait(t: float) -> None:
    print('Waiting...')
    time.sleep(t)

if wait(2) > 1:   # Error: can't compare None and int
    ...
```

The `Any` type is discussed in more detail in section *Dynamically typed code*.

Note: A function without any types in the signature is dynamically typed. The body of a dynamically typed function is not checked statically, and local variables have implicit `Any` types. This makes it easier to migrate legacy Python code to mypy, as mypy won't complain about dynamically typed functions.

9.3 Tuple types

The type `Tuple[T1, ..., Tn]` represents a tuple with the item types `T1`, ..., `Tn`:

```
def f(t: Tuple[int, str]) -> None:
    t = 1, 'foo'     # OK
    t = 'foo', 1     # Type check error
```

A tuple type of this kind has exactly a specific number of items (2 in the above example). Tuples can also be used as immutable, varying-length sequences. You can use the type `Tuple[T, ...]` (with a literal `...` – it's part of the syntax) for this purpose. Example:

```
def print_squared(t: Tuple[int, ...]) -> None:
    for n in t:
        print(n, n ** 2)

print_squared(())           # OK
print_squared((1, 3, 5))    # OK
print_squared([1, 2])       # Error: only a tuple is valid
```

Note: Usually it's a better idea to use `Sequence[T]` instead of `Tuple[T, ...]`, as `Sequence` is also compatible with lists and other non-tuple sequences.

Note: `Tuple[...]` is not valid as a base class outside stub files. This is a limitation of the `typing` module. One way to work around this is to use a named tuple as a base class (see section *Named tuples*).

9.4 Callable types (and lambdas)

You can pass around function objects and bound methods in statically typed code. The type of a function that accepts arguments `A1`, ..., `An` and returns `Rt` is `Callable[[A1, ..., An], Rt]`. Example:

```
from typing import Callable

def twice(i: int, next: Callable[[int], int]) -> int:
    return next(next(i))

def add(i: int) -> int:
    return i + 1

print(twice(3, add))    # 5
```

You can only have positional arguments, and only ones without default values, in callable types. These cover the vast majority of uses of callable types, but sometimes this isn't quite enough. Mypy recognizes a special form `Callable[..., T]` (with a literal `...`) which can be used in less typical cases. It is compatible with arbitrary callable objects that return a type compatible with `T`, independent of the number, types or kinds of arguments. Mypy lets you call such callable values with arbitrary arguments, without any checking – in this respect they are treated similar to a `(*args: Any, **kwargs: Any)` function signature. Example:

```
from typing import Callable

 def arbitrary_call(f: Callable[..., int]) -> int:
     return f('x') + f(y=2)  # OK

 arbitrary_call(ord)    # No static error, but fails at runtime
 arbitrary_call(open)   # Error: does not return an int
 arbitrary_call(1)      # Error: 'int' is not callable
```

Lambdas are also supported. The lambda argument and return value types cannot be given explicitly; they are always inferred based on context using bidirectional type inference:

```
l = map(lambda x: x + 1, [1, 2, 3])    # Infer x as int and l as List[int]
```

If you want to give the argument or return value types explicitly, use an ordinary, perhaps nested function definition.

9.5 Extended Callable types

As an experimental mypy extension, you can specify `Callable` types that support keyword arguments, optional arguments, and more. Where you specify the arguments of a Callable, you can choose to supply just the type of a nameless positional argument, or an "argument specifier" representing a more complicated form of argument. This allows one to more closely emulate the full range of possibilities given by the `def` statement in Python.

As an example, here's a complicated function definition and the corresponding `Callable`:

```
from typing import Callable
from mypy_extensions import (Arg, DefaultArg, NamedArg,
                             DefaultNamedArg, VarArg, KwArg)

def func(__a: int,  # This convention is for nameless arguments
         b: int,
         c: int = 0,
         *args: int,
         d: int,
         e: int = 0,
         **kwargs: int) -> int:
    ...

F = Callable[[int,  # Or Arg(int)
              Arg(int, 'b'),
              DefaultArg(int, 'c'),
              VarArg(int),
              NamedArg(int, 'd'),
              DefaultNamedArg(int, 'e'),
              KwArg(int)],
             int]

f: F = func
```

Argument specifiers are special function calls that can specify the following aspects of an argument:

- its type (the only thing that the basic format supports)
- its name (if it has one)
- whether it may be omitted
- whether it may or must be passed using a keyword
- whether it is a `*args` argument (representing the remaining positional arguments)
- whether it is a `**kwargs` argument (representing the remaining keyword arguments)

The following functions are available in `mypy_extensions` for this purpose:

```
def Arg(type=Any, name=None):
    # A normal, mandatory, positional argument.
    # If the name is specified it may be passed as a keyword.

def DefaultArg(type=Any, name=None):
    # An optional positional argument (i.e. with a default value).
    # If the name is specified it may be passed as a keyword.

def NamedArg(type=Any, name=None):
    # A mandatory keyword-only argument.
```

(continues on next page)

(continued from previous page)

```
def DefaultNamedArg(type=Any, name=None):
    # An optional keyword-only argument (i.e. with a default value).

def VarArg(type=Any):
    # A *args-style variadic positional argument.
    # A single VarArg() specifier represents all remaining
    # positional arguments.

def KwArg(type=Any):
    # A **kwargs-style variadic keyword argument.
    # A single KwArg() specifier represents all remaining
    # keyword arguments.
```

In all cases, the `type` argument defaults to `Any`, and if the `name` argument is omitted the argument has no name (the name is required for `NamedArg` and `DefaultNamedArg`). A basic `Callable` such as

```
MyFunc = Callable[[int, str, int], float]
```

is equivalent to the following:

```
MyFunc = Callable[[Arg(int), Arg(str), Arg(int)], float]
```

A `Callable` with unspecified argument types, such as

```
MyOtherFunc = Callable[..., int]
```

is (roughly) equivalent to

```
MyOtherFunc = Callable[[VarArg(), KwArg()], int]
```

Note: This feature is experimental. Details of the implementation may change and there may be unknown limitations. **IMPORTANT:** Each of the functions above currently just returns its `type` argument, so the information contained in the argument specifiers is not available at runtime. This limitation is necessary for backwards compatibility with the existing `typing.py` module as present in the Python 3.5+ standard library and distributed via PyPI.

9.6 Union types

Python functions often accept values of two or more different types. You can use overloading to model this in statically typed code, but union types can make code like this easier to write.

Use the `Union[T1, ..., Tn]` type constructor to construct a union type. For example, the type `Union[int, str]` is compatible with both integers and strings. You can use an `isinstance()` check to narrow down the type to a specific type:

```
from typing import Union

def f(x: Union[int, str]) -> None:
    x + 1     # Error: str + int is not valid
    if isinstance(x, int):
        # Here type of x is int.
        x + 1      # OK
    else:
```

(continues on next page)

(continued from previous page)

```
        # Here type of x is str.
        x + 'a'     # OK

f(1)    # OK
f('x')  # OK
f(1.1)  # Error
```

9.7 Optional types and the None type

You can use the `Optional` type modifier to define a type variant that allows `None`, such as `Optional[int]` (`Optional[X]` is the preferred shorthand for `Union[X, None]`):

```
from typing import Optional

def strlen(s: str) -> Optional[int]:
    if not s:
        return None  # OK
    return len(s)

def strlen_invalid(s: str) -> int:
    if not s:
        return None  # Error: None not compatible with int
    return len(s)
```

Most operations will not be allowed on unguarded `None` or `Optional` values:

```
def my_inc(x: Optional[int]) -> int:
    return x + 1  # Error: Cannot add None and int
```

Instead, an explicit `None` check is required. Mypy has powerful type inference that lets you use regular Python idioms to guard against `None` values. For example, mypy recognizes `is None` checks:

```
def my_inc(x: Optional[int]) -> int:
    if x is None:
        return 0
    else:
        # The inferred type of x is just int here.
        return x + 1
```

Mypy will infer the type of `x` to be `int` in the else block due to the check against `None` in the if condition.

Other supported checks for guarding against a `None` value include `if x is not None`, `if x` and `if not x`. Additionally, mypy understands `None` checks within logical expressions:

```
def concat(x: Optional[str], y: Optional[str]) -> Optional[str]:
    if x is not None and y is not None:
        # Both x and y are not None here
        return x + y
    else:
        return None
```

Sometimes mypy doesn't realize that a value is never `None`. This notably happens when a class instance can exist in a partially defined state, where some attribute is initialized to `None` during object construction, but a method assumes that the attribute is no longer `None`. Mypy will complain about the possible `None` value. You can use `assert x is not None` to work around this in the method:

```
class Resource:
    path: Optional[str] = None

    def initialize(self, path: str) -> None:
        self.path = path

    def read(self) -> str:
        # We require that the object has been initialized.
        assert self.path is not None
        with open(self.path) as f:  # OK
           return f.read()

r = Resource()
r.initialize('/foo/bar')
r.read()
```

When initializing a variable as `None`, `None` is usually an empty place-holder value, and the actual value has a different type. This is why you need to annotate an attribute in a cases like the class `Resource` above:

```
class Resource:
    path: Optional[str] = None
    ...
```

This also works for attributes defined within methods:

```
class Counter:
    def __init__(self) -> None:
        self.count: Optional[int] = None
```

As a special case, you can use a non-optional type when initializing an attribute to `None` inside a class body *and* using a type comment, since when using a type comment, an initializer is syntacticaly required, and `None` is used as a dummy, placeholder initializer:

```
from typing import List

class Container:
    items = None  # type: List[str]  # OK (only with type comment)
```

This is not a problem when using variable annotations, since no initializer is needed:

```
from typing import List

class Container:
    items: List[str]  # No initializer
```

Mypy generally uses the first assignment to a variable to infer the type of the variable. However, if you assign both a `None` value and a non-`None` value in the same scope, mypy can usually do the right thing without an annotation:

```
def f(i: int) -> None:
    n = None  # Inferred type Optional[int] because of the assignment below
    if i > 0:
        n = i
    ...
```

Sometimes you may get the error "Cannot determine type of <something>". In this case you should add an explicit `Optional[...]` annotation (or type comment).

Note: `None` is a type with only one value, `None`. `None` is also used as the return type for functions that don't return a value, i.e. functions that implicitly return `None`.

Note: The Python interpreter internally uses the name `NoneType` for the type of `None`, but `None` is always used in type annotations. The latter is shorter and reads better. (Besides, `NoneType` is not even defined in the standard library.)

Note: `Optional[...]` *does not* mean a function argument with a default value. However, if the default value of an argument is `None`, you can use an optional type for the argument, but it's not enforced by default. You can use the `--no-implicit-optional` command-line option to stop treating arguments with a `None` default value as having an implicit `Optional[...]` type. It's possible that this will become the default behavior in the future.

9.8 Disabling strict optional checking

Mypy also has an option to treat `None` as a valid value for every type (in case you know Java, it's useful to think of it as similar to the Java `null`). In this mode `None` is also valid for primitive types such as `int` and `float`, and `Optional[...]` types are not required.

The mode is enabled through the `--no-strict-optional` command-line option. In mypy versions before 0.600 this was the default mode. You can enable this option explicitly for backward compatibility with earlier mypy versions, in case you don't want to introduce optional types to your codebase yet.

It will cause mypy to silently accept some buggy code, such as this example – it's not recommended if you can avoid it:

```
def inc(x: int) -> int:
    return x + 1

x = inc(None)  # No error reported by mypy if strict optional mode disabled!
```

However, making code "optional clean" can take some work! You can also use *the mypy configuration file* to migrate your code to strict optional checking one file at a time, since there exists the *per-module flag* `strict_optional` to control strict optional mode.

Often it's still useful to document whether a variable can be `None`. For example, this function accepts a `None` argument, but it's not obvious from its signature:

```
def greeting(name: str) -> str:
    if name:
        return 'Hello, {}'.format(name)
    else:
        return 'Hello, stranger'

print(greeting('Python'))  # Okay!
print(greeting(None))      # Also okay!
```

You can still use `Optional[t]` to document that `None` is a valid argument type, even if strict `None` checking is not enabled:

```
from typing import Optional

def greeting(name: Optional[str]) -> str:
    if name:
        return 'Hello, {}'.format(name)
    else:
        return 'Hello, stranger'
```

Mypy treats this as semantically equivalent to the previous example if strict optional checking is disabled, since `None` is implicitly valid for any type, but it's much more useful for a programmer who is reading the code. This also makes it easier to migrate to strict `None` checking in the future.

9.9 The NoReturn type

Mypy provides support for functions that never return. For example, a function that unconditionally raises an exception:

```
from mypy_extensions import NoReturn

def stop() -> NoReturn:
    raise Exception('no way')
```

Mypy will ensure that functions annotated as returning `NoReturn` truly never return, either implicitly or explicitly. Mypy will also recognize that the code after calls to such functions is unreachable and will behave accordingly:

```
def f(x: int) -> int:
    if x == 0:
        return x
    stop()
    return 'whatever works'  # No error in an unreachable block
```

Install `mypy_extensions` using pip to use `NoReturn` in your code. Python 3 command line:

```
python3 -m pip install --upgrade mypy-extensions
```

This works for Python 2:

```
pip install --upgrade mypy-extensions
```

9.10 Class name forward references

Python does not allow references to a class object before the class is defined. Thus this code does not work as expected:

```
def f(x: A) -> None:  # Error: Name A not defined
    ....

class A:
    ...
```

In cases like these you can enter the type as a string literal — this is a *forward reference*:

```
def f(x: 'A') -> None:  # OK
    ...

class A:
    ...
```

Of course, instead of using a string literal type, you could move the function definition after the class definition. This is not always desirable or even possible, though.

Any type can be entered as a string literal, and you can combine string-literal types with non-string-literal types freely:

```
def f(a: List['A']) -> None: ...  # OK
def g(n: 'int') -> None: ...      # OK, though not useful

class A: pass
```

String literal types are never needed in `# type:` comments.

String literal types must be defined (or imported) later *in the same module*. They cannot be used to leave cross-module references unresolved. (For dealing with import cycles, see *Import cycles*.)

9.11 Type aliases

In certain situations, type names may end up being long and painful to type:

```
def f() -> Union[List[Dict[Tuple[int, str], Set[int]]], Tuple[str, List[str]]]:
    ...
```

When cases like this arise, you can define a type alias by simply assigning the type to a variable:

```
AliasType = Union[List[Dict[Tuple[int, str], Set[int]]], Tuple[str, List[str]]]

# Now we can use AliasType in place of the full name:

def f() -> AliasType:
    ...
```

Type aliases can be generic, in this case they could be used in two variants: Subscripted aliases are equivalent to original types with substituted type variables, number of type arguments must match the number of free type variables in generic type alias. Unsubscripted aliases are treated as original types with free variables replaced with `Any`. Examples (following PEP 484):

```
from typing import TypeVar, Iterable, Tuple, Union, Callable
S = TypeVar('S')
TInt = Tuple[int, S]
UInt = Union[S, int]
CBack = Callable[..., S]

def response(query: str) -> UInt[str]:  # Same as Union[str, int]
    ...
def activate(cb: CBack[S]) -> S:        # Same as Callable[..., S]
    ...
table_entry: TInt  # Same as Tuple[int, Any]

T = TypeVar('T', int, float, complex)
```

(continues on next page)

(continued from previous page)

```
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T:
    return sum(x*y for x, y in v)

def dilate(v: Vec[T], scale: T) -> Vec[T]:
    return ((x * scale, y * scale) for x, y in v)

v1: Vec[int] = []       # Same as Iterable[Tuple[int, int]]
v2: Vec = []            # Same as Iterable[Tuple[Any, Any]]
v3: Vec[int, int] = [] # Error: Invalid alias, too many type arguments!
```

Type aliases can be imported from modules like any names. Aliases can target another aliases (although building complex chains of aliases is not recommended, this impedes code readability, thus defeating the purpose of using aliases). Following previous examples:

```
from typing import TypeVar, Generic, Optional
from first_example import AliasType
from second_example import Vec

def fun() -> AliasType:
    ...

T = TypeVar('T')
class NewVec(Generic[T], Vec[T]):
    ...
for i, j in NewVec[int]():
    ...

OIntVec = Optional[Vec[int]]
```

Note: A type alias does not create a new type. It's just a shorthand notation for another type – it's equivalent to the target type. For generic type aliases this means that variance of type variables used for alias definition does not apply to aliases. A parameterized generic alias is treated simply as an original type with the corresponding type variables substituted.

9.12 NewTypes

(Freely after PEP 484.)

There are also situations where a programmer might want to avoid logical errors by creating simple classes. For example:

```
class UserId(int):
    pass

get_by_user_id(user_id: UserId):
    ...
```

However, this approach introduces some runtime overhead. To avoid this, the typing module provides a helper function `NewType` that creates simple unique types with almost zero runtime overhead. Mypy will treat the statement `Derived = NewType('Derived', Base)` as being roughly equivalent to the following definition:

```
class Derived(Base):
    def __init__(self, _x: Base) -> None:
        ...
```

However, at runtime, `NewType('Derived', Base)` will return a dummy function that simply returns its argument:

```
def Derived(_x):
    return _x
```

Mypy will require explicit casts from `int` where `UserId` is expected, while implicitly casting from `UserId` where `int` is expected. Examples:

```
from typing import NewType

UserId = NewType('UserId', int)

def name_by_id(user_id: UserId) -> str:
    ...

UserId('user')          # Fails type check

name_by_id(42)          # Fails type check
name_by_id(UserId(42))  # OK

num = UserId(5) + 1     # type: int
```

`NewType` accepts exactly two arguments. The first argument must be a string literal containing the name of the new type and must equal the name of the variable to which the new type is assigned. The second argument must be a properly subclassable class, i.e., not a type construct like `Union`, etc.

The function returned by `NewType` accepts only one argument; this is equivalent to supporting only one constructor accepting an instance of the base class (see above). Example:

```
from typing import NewType

class PacketId:
    def __init__(self, major: int, minor: int) -> None:
        self._major = major
        self._minor = minor

TcpPacketId = NewType('TcpPacketId', PacketId)

packet = PacketId(100, 100)
tcp_packet = TcpPacketId(packet)  # OK

tcp_packet = TcpPacketId(127, 0)  # Fails in type checker and at runtime
```

Both `isinstance` and `issubclass`, as well as subclassing will fail for `NewType('Derived', Base)` since function objects don't support these operations.

Note: Note that unlike type aliases, `NewType` will create an entirely new and unique type when used. The intended purpose of `NewType` is to help you detect cases where you accidentally mixed together the old base type and the new derived type.

For example, the following will successfully typecheck when using type aliases:

```
UserId = int

def name_by_id(user_id: UserId) -> str:
    ...

name_by_id(3)  # ints and UserId are synonymous
```

But a similar example using `NewType` will not typecheck:

```
from typing import NewType

UserId = NewType('UserId', int)

def name_by_id(user_id: UserId) -> str:
    ...

name_by_id(3)  # int is not the same as UserId
```

9.13 Named tuples

Mypy recognizes named tuples and can type check code that defines or uses them. In this example, we can detect code trying to access a missing attribute:

```
Point = namedtuple('Point', ['x', 'y'])
p = Point(x=1, y=2)
print(p.z)  # Error: Point has no attribute 'z'
```

If you use `namedtuple` to define your named tuple, all the items are assumed to have `Any` types. That is, mypy doesn't know anything about item types. You can use `typing.NamedTuple` to also define item types:

```
from typing import NamedTuple

Point = NamedTuple('Point', [('x', int),
                             ('y', int)])
p = Point(x=1, y='x')  # Argument has incompatible type "str"; expected "int"
```

Python 3.6 will have an alternative, class-based syntax for named tuples with types. Mypy supports it already:

```
from typing import NamedTuple

class Point(NamedTuple):
    x: int
    y: int

p = Point(x=1, y='x')  # Argument has incompatible type "str"; expected "int"
```

9.14 The type of class objects

(Freely after PEP 484.)

Sometimes you want to talk about class objects that inherit from a given class. This can be spelled as `Type[C]` where `C` is a class. In other words, when `C` is the name of a class, using `C` to annotate an argument declares that the argument

is an instance of `C` (or of a subclass of `C`), but using `Type[C]` as an argument annotation declares that the argument is a class object deriving from `C` (or `C` itself).

For example, assume the following classes:

```
class User:
    # Defines fields like name, email

class BasicUser(User):
    def upgrade(self):
        """Upgrade to Pro"""

class ProUser(User):
    def pay(self):
        """Pay bill"""
```

Note that `ProUser` doesn't inherit from `BasicUser`.

Here's a function that creates an instance of one of these classes if you pass it the right class object:

```
def new_user(user_class):
    user = user_class()
    # (Here we could write the user object to a database)
    return user
```

How would we annotate this function? Without `Type[]` the best we could do would be:

```
def new_user(user_class: type) -> User:
    # Same  implementation as before
```

This seems reasonable, except that in the following example, mypy doesn't see that the `buyer` variable has type `ProUser`:

```
buyer = new_user(ProUser)
buyer.pay()  # Rejected, not a method on User
```

However, using `Type[]` and a type variable with an upper bound (see *Type variables with upper bounds*) we can do better:

```
U = TypeVar('U', bound=User)

def new_user(user_class: Type[U]) -> U:
    # Same  implementation as before
```

Now mypy will infer the correct type of the result when we call `new_user()` with a specific subclass of `User`:

```
beginner = new_user(BasicUser)  # Inferred type is BasicUser
beginner.upgrade()  # OK
```

Note: The value corresponding to `Type[C]` must be an actual class object that's a subtype of `C`. Its constructor must be compatible with the constructor of `C`. If `C` is a type variable, its upper bound must be a class object.

For more details about `Type[]` see PEP 484.

9.15 Text and AnyStr

Sometimes you may want to write a function which will accept only unicode strings. This can be challenging to do in a codebase intended to run in both Python 2 and Python 3 since `str` means something different in both versions and `unicode` is not a keyword in Python 3.

To help solve this issue, use `typing.Text` which is aliased to `unicode` in Python 2 and to `str` in Python 3. This allows you to indicate that a function should accept only unicode strings in a cross-compatible way:

```
from typing import Text

def unicode_only(s: Text) -> Text:
    return s + u'\u2713'
```

In other cases, you may want to write a function that will work with any kind of string but will not let you mix two different string types. To do so use `typing.AnyStr`:

```
from typing import AnyStr

def concat(x: AnyStr, y: AnyStr) -> AnyStr:
    return x + y

concat('a', 'b')      # Okay
concat(b'a', b'b')    # Okay
concat('a', b'b')     # Error: cannot mix bytes and unicode
```

For more details, see *Type variables with value restriction*.

Note: How `bytes`, `str`, and `unicode` are handled between Python 2 and Python 3 may change in future versions of mypy.

9.16 Generators

A basic generator that only yields values can be annotated as having a return type of either `Iterator[YieldType]` or `Iterable[YieldType]`. For example:

```
def squares(n: int) -> Iterator[int]:
    for i in range(n):
        yield i * i
```

If you want your generator to accept values via the `send` method or return a value, you should use the `Generator[YieldType, SendType, ReturnType]` generic type instead. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generics in the typing module, the `SendType` of `Generator` behaves contravariantly, not covariantly or invariantly.

If you do not plan on receiving or returning values, then set the `SendType` or `ReturnType` to `None`, as appropriate. For example, we could have annotated the first example as the following:

```
def squares(n: int) -> Generator[int, None, None]:
    for i in range(n):
        yield i * i
```

This is slightly different from using `Iterable[int]` or `Iterator[int]`, since generators have `close()`, `send()`, and `throw()` methods that generic iterables don't. If you will call these methods on the returned generator, use the `Generator` type instead of `Iterable` or `Iterator`.

9.17 Typing async/await

Mypy supports the ability to type coroutines that use the `async/await` syntax introduced in Python 3.5. For more information regarding coroutines and this new syntax, see PEP 492.

Functions defined using `async def` are typed just like normal functions. The return type annotation should be the same as the type of the value you expect to get back when `await`-ing the coroutine.

```
import asyncio

async def format_string(tag: str, count: int) -> str:
    return 'T-minus {} ({})'.format(count, tag)

async def countdown_1(tag: str, count: int) -> str:
    while count > 0:
        my_str = await format_string(tag, count)  # has type 'str'
        print(my_str)
        await asyncio.sleep(0.1)
        count -= 1
    return "Blastoff!"

loop = asyncio.get_event_loop()
loop.run_until_complete(countdown_1("Millennium Falcon", 5))
loop.close()
```

The result of calling an `async def` function *without awaiting* will be a value of type `Awaitable[T]`:

```
my_coroutine = countdown_1("Millennium Falcon", 5)
reveal_type(my_coroutine)  # has type 'Awaitable[str]'
```

Note: *reveal_type()* displays the inferred static type of an expression.

If you want to use coroutines in older versions of Python that do not support the `async def` syntax, you can instead use the `@asyncio.coroutine` decorator to convert a generator into a coroutine.

Note that we set the `YieldType` of the generator to be `Any` in the following example. This is because the exact yield type is an implementation detail of the coroutine runner (e.g. the `asyncio` event loop) and your coroutine shouldn't have to know or care about what precisely that type is.

```
from typing import Any, Generator
import asyncio

@asyncio.coroutine
def countdown_2(tag: str, count: int) -> Generator[Any, None, str]:
    while count > 0:
```

(continues on next page)

(continued from previous page)

```
        print('T-minus {} ({})'.format(count, tag))
        yield from asyncio.sleep(0.1)
        count -= 1
    return "Blastoff!"

loop = asyncio.get_event_loop()
loop.run_until_complete(countdown_2("USS Enterprise", 5))
loop.close()
```

As before, the result of calling a generator decorated with `@asyncio.coroutine` will be a value of type `Awaitable[T]`.

Note: At runtime, you are allowed to add the `@asyncio.coroutine` decorator to both functions and generators. This is useful when you want to mark a work-in-progress function as a coroutine, but have not yet added `yield` or `yield from` statements:

```
import asyncio

@asyncio.coroutine
def serialize(obj: object) -> str:
    # todo: add yield/yield from to turn this into a generator
    return "placeholder"
```

However, mypy currently does not support converting functions into coroutines. Support for this feature will be added in a future version, but for now, you can manually force the function to be a generator by doing something like this:

```
from typing import Generator
import asyncio

@asyncio.coroutine
def serialize(obj: object) -> Generator[None, None, str]:
    # todo: add yield/yield from to turn this into a generator
    if False:
        yield
    return "placeholder"
```

You may also choose to create a subclass of `Awaitable` instead:

```
from typing import Any, Awaitable, Generator
import asyncio

class MyAwaitable(Awaitable[str]):
    def __init__(self, tag: str, count: int) -> None:
        self.tag = tag
        self.count = count

    def __await__(self) -> Generator[Any, None, str]:
        for i in range(n, 0, -1):
            print('T-minus {} ({})'.format(i, tag))
            yield from asyncio.sleep(0.1)
        return "Blastoff!"

def countdown_3(tag: str, count: int) -> Awaitable[str]:
    return MyAwaitable(tag, count)
```

(continues on next page)

(continued from previous page)

```
loop = asyncio.get_event_loop()
loop.run_until_complete(countdown_3("Heart of Gold", 5))
loop.close()
```

To create an iterable coroutine, subclass `AsyncIterator`:

```
from typing import Optional, AsyncIterator
import asyncio

class arange(AsyncIterator[int]):
    def __init__(self, start: int, stop: int, step: int) -> None:
        self.start = start
        self.stop = stop
        self.step = step
        self.count = start - step

    def __aiter__(self) -> AsyncIterator[int]:
        return self

    async def __anext__(self) -> int:
        self.count += self.step
        if self.count == self.stop:
            raise StopAsyncIteration
        else:
            return self.count

async def countdown_4(tag: str, n: int) -> str:
    async for i in arange(n, 0, -1):
        print('T-minus {} ({})'.format(i, tag))
        await asyncio.sleep(0.1)
    return "Blastoff!"

loop = asyncio.get_event_loop()
loop.run_until_complete(countdown_4("Serenity", 5))
loop.close()
```

For a more concrete example, the mypy repo has a toy webcrawler that demonstrates how to work with coroutines. One version uses async/await and one uses yield from.

9.18 TypedDict

Note: TypedDict is an officially supported feature, but it is still experimental.

Python programs often use dictionaries with string keys to represent objects. Here is a typical example:

```
movie = {'name': 'Blade Runner', 'year': 1982}
```

Only a fixed set of string keys is expected (`'name'` and `'year'` above), and each key has an independent value type (`str` for `'name'` and `int` for `'year'` above). We've previously seen the `Dict[K, V]` type, which lets you declare uniform dictionary types, where every value has the same type, and arbitrary keys are supported. This is clearly not a good fit for `movie` above. Instead, you can use a `TypedDict` to give a precise type for objects like `movie`, where the type of each dictionary value depends on the key:

```
from mypy_extensions import TypedDict

Movie = TypedDict('Movie', {'name': str, 'year': int})

movie = {'name': 'Blade Runner', 'year': 1982}  # type: Movie
```

`Movie` is a TypedDict type with two items: `'name'` (with type `str`) and `'year'` (with type `int`). Note that we used an explicit type annotation for the `movie` variable. This type annotation is important – without it, mypy will try to infer a regular, uniform `Dict` type for `movie`, which is not what we want here.

Note: If you pass a TypedDict object as an argument to a function, no type annotation is usually necessary since mypy can infer the desired type based on the declared argument type. Also, if an assignment target has been previously defined, and it has a TypedDict type, mypy will treat the assigned value as a TypedDict, not `Dict`.

Now mypy will recognize these as valid:

```
name = movie['name']  # Okay; type of name is str
year = movie['year']  # Okay; type of year is int
```

Mypy will detect an invalid key as an error:

```
director = movie['director']  # Error: 'director' is not a valid key
```

Mypy will also reject a runtime-computed expression as a key, as it can't verify that it's a valid key. You can only use string literals as TypedDict keys.

The `TypedDict` type object can also act as a constructor. It returns a normal `dict` object at runtime – a `TypedDict` does not define a new runtime type:

```
toy_story = Movie(name='Toy Story', year=1995)
```

This is equivalent to just constructing a dictionary directly using `{ ... }` or `dict(key=value, ...)`. The constructor form is sometimes convenient, since it can be used without a type annotation, and it also makes the type of the object explicit.

Like all types, TypedDicts can be used as components to build arbitrarily complex types. For example, you can define nested TypedDicts and containers with TypedDict items. Unlike most other types, mypy uses structural compatibility checking (or structural subtyping) with TypedDicts. A TypedDict object with extra items is compatible with a narrower TypedDict, assuming item types are compatible (*totality* also affects subtyping, as discussed below).

Note: You need to install `mypy_extensions` using pip to use `TypedDict`:

```
python3 -m pip install --upgrade mypy-extensions
```

Or, if you are using Python 2:

```
pip install --upgrade mypy-extensions
```

9.18.1 Totality

By default mypy ensures that a TypedDict object has all the specified keys. This will be flagged as an error:

```
# Error: 'year' missing
toy_story = {'name': 'Toy Story'}  # type: Movie
```

Sometimes you want to allow keys to be left out when creating a TypedDict object. You can provide the `total=False` argument to `TypedDict(...)` to achieve this:

```
GuiOptions = TypedDict(
    'GuiOptions', {'language': str, 'color': str}, total=False)
options = {}  # type: GuiOptions  # Okay
options['language'] = 'en'
```

You may need to use `get()` to access items of a partial (non-total) TypedDict, since indexing using `[]` could fail at runtime. However, mypy still lets use `[]` with a partial TypedDict – you just need to be careful with it, as it could result in a `KeyError`. Requiring `get()` everywhere would be too cumbersome. (Note that you are free to use `get()` with total TypedDicts as well.)

Keys that aren't required are shown with a `?` in error messages:

```
# Revealed type is 'TypedDict('GuiOptions', {'language'?: builtins.str,
#                                            'color'?: builtins.str})'
reveal_type(options)
```

Totality also affects structural compatibility. You can't use a partial TypedDict when a total one is expected. Also, a total typed dict is not valid when a partial one is expected.

9.18.2 Class-based syntax

Python 3.6 supports an alternative, class-based syntax to define a TypedDict. This means that your code must be checked as if it were Python 3.6 (using the `--python-version` flag on the command line, for example). Simply running mypy on Python 3.6 is insufficient.

```
from mypy_extensions import TypedDict

class Movie(TypedDict):
    name: str
    year: int
```

The above definition is equivalent to the original `Movie` definition. It doesn't actually define a real class. This syntax also supports a form of inheritance – subclasses can define additional items. However, this is primarily a notational shortcut. Since mypy uses structural compatibility with TypedDicts, inheritance is not required for compatibility. Here is an example of inheritance:

```
class Movie(TypedDict):
    name: str
    year: int

class BookBasedMovie(Movie):
    based_on: str
```

Now `BookBasedMovie` has keys `name`, `year` and `based_on`.

9.18.3 Mixing required and non-required items

In addition to allowing reuse across TypedDict types, inheritance also allows you to mix required and non-required (using `total=False`) items in a single TypedDict. Example:

```
class MovieBase(TypedDict):
    name: str
    year: int

class Movie(MovieBase, total=False):
    based_on: str
```

Now `Movie` has required keys `name` and `year`, while `based_on` can be left out when constructing an object. A TypedDict with a mix of required and non-required keys, such as `Movie` above, will only be compatible with another TypedDict if all required keys in the other TypedDict are required keys in the first TypedDict, and all non-required keys of the other TypedDict are also non-required keys in the first TypedDict.

CHAPTER 10

Class basics

10.1 Instance and class attributes

Mypy type checker detects if you are trying to access a missing attribute, which is a very common programming error. For this to work correctly, instance and class attributes must be defined or initialized within the class. Mypy infers the types of attributes:

```
class A:
    def __init__(self, x: int) -> None:
        self.x = x      # Attribute x of type int

a = A(1)
a.x = 2         # OK
a.y = 3         # Error: A has no attribute y
```

This is a bit like each class having an implicitly defined `__slots__` attribute. This is only enforced during type checking and not when your program is running.

You can declare types of variables in the class body explicitly using a type comment:

```
class A:
    x = None  # type: List[int]  # Declare attribute x of type List[int]

a = A()
a.x = [1]      # OK
```

As in Python, a variable defined in the class body can used as a class or an instance variable.

Similarly, you can give explicit types to instance variables defined in a method:

```
class A:
    def __init__(self) -> None:
        self.x = []  # type: List[int]
```

(continues on next page)

(continued from previous page)

```
    def f(self) -> None:
        self.y = 0  # type: Any
```

You can only define an instance variable within a method if you assign to it explicitly using `self`:

```
class A:
    def __init__(self) -> None:
        self.y = 1   # Define y
        a = self
        a.x = 1      # Error: x not defined
```

10.2 Overriding statically typed methods

When overriding a statically typed method, mypy checks that the override has a compatible signature:

```
class A:
    def f(self, x: int) -> None:
        ...

class B(A):
    def f(self, x: str) -> None:    # Error: type of x incompatible
        ...

class C(A):
    def f(self, x: int, y: int) -> None:  # Error: too many arguments
        ...

class D(A):
    def f(self, x: int) -> None:    # OK
        ...
```

Note: You can also vary return types **covariantly** in overriding. For example, you could override the return type `object` with a subtype such as `int`.

You can also override a statically typed method with a dynamically typed one. This allows dynamically typed code to override methods defined in library classes without worrying about their type signatures.

There is no runtime enforcement that the method override returns a value that is compatible with the original return type, since annotations have no effect at runtime:

```
class A:
    def inc(self, x: int) -> int:
        return x + 1

class B(A):
    def inc(self, x):       # Override, dynamically typed
        return 'hello'

b = B()
print(b.inc(1))    # hello
a = b # type: A
print(a.inc(1))    # hello
```

10.3 Abstract base classes and multiple inheritance

Mypy supports Python abstract base classes (ABCs). Abstract classes have at least one abstract method or property that must be implemented by a subclass. You can define abstract base classes using the `abc.ABCMeta` metaclass, and the `abc.abstractmethod` and `abc.abstractproperty` function decorators. Example:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @abstractmethod
    def foo(self, x: int) -> None: pass

    @abstractmethod
    def bar(self) -> str: pass

class B(A):
    def foo(self, x: int) -> None: ...
    def bar(self) -> str:
        return 'x'

a = A()  # Error: A is abstract
b = B()  # OK
```

Note that mypy performs checking for unimplemented abstract methods even if you omit the `ABCMeta` metaclass. This can be useful if the metaclass would cause runtime metaclass conflicts.

A class can inherit any number of classes, both abstract and concrete. As with normal overrides, a dynamically typed method can implement a statically typed method defined in any base class, including an abstract method defined in an abstract base class.

You can implement an abstract property using either a normal property or an instance variable.

10.4 Protocols and structural subtyping

Mypy supports two ways of deciding whether two classes are compatible as types: nominal subtyping and structural subtyping. *Nominal* subtyping is strictly based on the class hierarchy. If class `D` inherits class `C`, it's also a subtype of `C`, and instances of `D` can be used when `C` instances are expected. This form of subtyping is used by default in mypy, since it's easy to understand and produces clear and concise error messages, and since it matches how the native `isinstance()` check works – based on class hierarchy. *Structural* subtyping can also be useful. Class `D` is a structural subtype of class `C` if the former has all attributes and methods of the latter, and with compatible types.

Structural subtyping can be seen as a static equivalent of duck typing, which is well known to Python programmers. Mypy provides support for structural subtyping via protocol classes described below. See PEP 544 for the detailed specification of protocols and structural subtyping in Python.

10.5 Predefined protocols

The `typing` module defines various protocol classes that correspond to common Python protocols, such as `Iterable[T]`. If a class defines a suitable `__iter__` method, mypy understands that it implements the iterable protocol and is compatible with `Iterable[T]`. For example, `IntList` below is iterable, over `int` values:

```
from typing import Iterator, Iterable, Optional

class IntList:
    def __init__(self, value: int, next: Optional[IntList]) -> None:
        self.value = value
        self.next = next

    def __iter__(self) -> Iterator[int]:
        current = self
        while current:
            yield current.value
            current = current.next

def print_numbered(items: Iterable[int]) -> None:
    for n, x in enumerate(items):
        print(n + 1, x)

x = IntList(3, IntList(5, None))
print_numbered(x)  # OK
print_numbered([4, 5])  # Also OK
```

The subsections below introduce all built-in protocols defined in `typing` and the signatures of the corresponding methods you need to define to implement each protocol (the signatures can be left out, as always, but mypy won't type check unannotated methods).

10.5.1 Iteration protocols

The iteration protocols are useful in many contexts. For example, they allow iteration of objects in for loops.

Iterable[T]

The *example above* has a simple implementation of an `__iter__` method.

```
def __iter__(self) -> Iterator[T]
```

Iterator[T]

```
def __next__(self) -> T
def __iter__(self) -> Iterator[T]
```

10.5.2 Collection protocols

Many of these are implemented by built-in container types such as `list` and `dict`, and these are also useful for user-defined collection objects.

Sized

This is a type for objects that support `len(x)`.

```
def __len__(self) -> int
```

Container[T]

This is a type for objects that support the `in` operator.

```
def __contains__(self, x: object) -> bool
```

Collection[T]

```
def __len__(self) -> int
def __iter__(self) -> Iterator[T]
def __contains__(self, x: object) -> bool
```

10.5.3 One-off protocols

These protocols are typically only useful with a single standard library function or class.

Reversible[T]

This is a type for objects that support `reversed(x)`.

```
def __reversed__(self) -> Iterator[T]
```

SupportsAbs[T]

This is a type for objects that support `abs(x)`. `T` is the type of value returned by `abs(x)`.

```
def __abs__(self) -> T
```

SupportsBytes

This is a type for objects that support `bytes(x)`.

```
def __bytes__(self) -> bytes
```

SupportsComplex

This is a type for objects that support `complex(x)`. Note that no arithmetic operations are supported.

```
def __complex__(self) -> complex
```

SupportsFloat

This is a type for objects that support `float(x)`. Note that no arithmetic operations are supported.

```
def __float__(self) -> float
```

SupportsInt

This is a type for objects that support `int(x)`. Note that no arithmetic operations are supported.

```
def __int__(self) -> int
```

SupportsRound[T]

This is a type for objects that support `round(x)`.

```
def __round__(self) -> T
```

10.5.4 Async protocols

These protocols can be useful in async code.

Awaitable[T]

```
def __await__(self) -> Generator[Any, None, T]
```

AsyncIterable[T]

```
def __aiter__(self) -> AsyncIterator[T]
```

AsyncIterator[T]

```
def __anext__(self) -> Awaitable[T]
def __aiter__(self) -> AsyncIterator[T]
```

10.5.5 Context manager protocols

There are two protocols for context managers – one for regular context managers and one for async ones. These allow defining objects that can be used in `with` and `async with` statements.

ContextManager[T]

```
def __enter__(self) -> T
def __exit__(self,
             exc_type: Optional[Type[BaseException]],
             exc_value: Optional[BaseException],
             traceback: Optional[TracebackType]) -> Optional[bool]
```

AsyncContextManager[T]

```
def __aenter__(self) -> Awaitable[T]
def __aexit__(self,
              exc_type: Optional[Type[BaseException]],
              exc_value: Optional[BaseException],
              traceback: Optional[TracebackType]) -> Awaitable[Optional[bool]]
```

10.6 Simple user-defined protocols

You can define your own protocol class by inheriting the special `typing_extensions.Protocol` class:

```
from typing import Iterable
from typing_extensions import Protocol

class SupportsClose(Protocol):
    def close(self) -> None:
        ...  # Explicit '...'

class Resource:  # No SupportsClose base class!
    # ... some methods ...

    def close(self) -> None:
        self.resource.release()

def close_all(items: Iterable[SupportsClose]) -> None:
    for item in items:
        item.close()

close_all([Resource(), open('some/file')])  # Okay!
```

`Resource` is a subtype of the `SupportClose` protocol since it defines a compatible `close` method. Regular file objects returned by `open()` are similarly compatible with the protocol, as they support `close()`.

Note: The `Protocol` base class is currently provided in the `typing_extensions` package. Once structural subtyping is mature and PEP 544 has been accepted, `Protocol` will be included in the `typing` module.

10.7 Defining subprotocols and subclassing protocols

You can also define subprotocols. Existing protocols can be extended and merged using multiple inheritance. Example:

```
# ... continuing from the previous example

class SupportsRead(Protocol):
    def read(self, amount: int) -> bytes: ...

class TaggedReadableResource(SupportsClose, SupportsRead, Protocol):
    label: str

class AdvancedResource(Resource):
    def __init__(self, label: str) -> None:
        self.label = label

    def read(self, amount: int) -> bytes:
        # some implementation
        ...

resource: TaggedReadableResource
resource = AdvancedResource('handle with care')  # OK
```

Note that inheriting from an existing protocol does not automatically turn the subclass into a protocol – it just creates a regular (non-protocol) class or ABC that implements the given protocol (or protocols). The `typing_extensions.Protocol` base class must always be explicitly present if you are defining a protocol:

```
class NewProtocol(SupportsClose):  # This is NOT a protocol
    new_attr: int

class Concrete:
   new_attr: int = 0

   def close(self) -> None:
       ...

# Error: nominal subtyping used by default
x: NewProtocol = Concrete()  # Error!
```

You can also include default implementations of methods in protocols. If you explicitly subclass these protocols you can inherit these default implementations. Explicitly including a protocol as a base class is also a way of documenting that your class implements a particular protocol, and it forces mypy to verify that your class implementation is actually compatible with the protocol.

Note: You can use Python 3.6 variable annotations (PEP 526) to declare protocol attributes. On Python 2.7 and earlier Python 3 versions you can use type comments and properties.

10.8 Recursive protocols

Protocols can be recursive (self-referential) and mutually recursive. This is useful for declaring abstract recursive collections such as trees and linked lists:

```
from typing import TypeVar, Optional
from typing_extensions import Protocol

class TreeLike(Protocol):
    value: int
```

(continues on next page)

(continued from previous page)

```
    @property
    def left(self) -> Optional['TreeLike']: ...

    @property
    def right(self) -> Optional['TreeLike']: ...

class SimpleTree:
    def __init__(self, value: int) -> None:
        self.value = value
        self.left: Optional['SimpleTree'] = None
        self.right: Optional['SimpleTree'] = None

root = SimpleTree(0)  # type: TreeLike  # OK
```

10.9 Using `isinstance()` with protocols

You can use a protocol class with `isinstance()` if you decorate it with the `typing_extensions.runtime` class decorator. The decorator adds support for basic runtime structural checks:

```
from typing_extensions import Protocol, runtime

@runtime
class Portable(Protocol):
    handles: int

class Mug:
    def __init__(self) -> None:
        self.handles = 1

mug = Mug()
if isinstance(mug, Portable):
   use(mug.handles)  # Works statically and at runtime
```

`isinstance()` also works with the *predefined protocols* in `typing` such as `Iterable`.

Note: `isinstance()` with protocols is not completely safe at runtime. For example, signatures of methods are not checked. The runtime implementation only checks that all protocol members are defined.

CHAPTER 11

Dynamically typed code

As mentioned earlier, bodies of functions that don't have have any explicit types in their function annotation are dynamically typed (operations are checked at runtime). Code outside functions is statically typed by default, and types of variables are inferred. This does usually the right thing, but you can also make any variable dynamically typed by defining it explicitly with the type `Any`:

```
from typing import Any

s = 1                 # Statically typed (type int)
d = 1  # type: Any    # Dynamically typed (type Any)
s = 'x'               # Type check error
d = 'x'               # OK
```

11.1 Operations on Any values

You can do anything using a value with type `Any`, and type checker does not complain:

```
def f(x: Any) -> int:
    # All of these are valid!
    x.foobar(1, y=2)
    print(x[3] + 'f')
    if x:
        x.z = x(2)
    open(x).read()
    return x
```

Values derived from an `Any` value also often have the type `Any` implicitly, as mypy can't infer a more precise result type. For example, if you get the attribute of an `Any` value or call a `Any` value the result is `Any`:

```
def f(x: Any) -> None:
    y = x.foo()  # y has type Any
    y.bar()      # Okay as well!
```

`Any` types may propagate through your program, making type checking less effective, unless you are careful.

11.2 Any vs. object

The type `object` is another type that can have an instance of arbitrary type as a value. Unlike `Any`, `object` is an ordinary static type (it is similar to `Object` in Java), and only operations valid for *all* types are accepted for `object` values. These are all valid:

```
def f(o: object) -> None:
    if o:
        print(o)
    print(isinstance(o, int))
    o = 2
    o = 'foo'
```

These are, however, flagged as errors, since not all objects support these operations:

```
def f(o: object) -> None:
    o.foo()        # Error!
    o + 2          # Error!
    open(o)        # Error!
    n = 1  # type: int
    n = o          # Error!
```

You can use `cast()` (see chapter *Casts*) or `isinstance` to go from a general type such as `object` to a more specific type (subtype) such as `int`. `cast()` is not needed with dynamically typed values (values with type `Any`).

CHAPTER 12

Function Overloading

Sometimes the types in a function depend on each other in ways that can't be captured with a `Union`. For example, the `__getitem__` (`[]` bracket indexing) method can take an integer and return a single item, or take a `slice` and return a `Sequence` of items. You might be tempted to annotate it like so:

```
from typing import Sequence, TypeVar, Union
T = TypeVar('T')

class MyList(Sequence[T]):
    def __getitem__(self, index: Union[int, slice]) -> Union[T, Sequence[T]]:
        if isinstance(index, int):
            ...  # Return a T here
        elif isinstance(index, slice):
            ...  # Return a sequence of Ts here
        else:
            raise TypeError(...)
```

But this is too loose, as it implies that when you pass in an `int` you might sometimes get out a single item and sometimes a sequence. The return type depends on the parameter type in a way that can't be expressed using a type variable. Instead, we can use overloading to give the same function multiple type annotations (signatures) and accurately describe the function's behavior.

```
from typing import overload, Sequence, TypeVar, Union
T = TypeVar('T')

class MyList(Sequence[T]):

    # The @overload definitions are just for the type checker,
    # and overwritten by the real implementation below.
    @overload
    def __getitem__(self, index: int) -> T:
        pass  # Don't put code here

    # All overloads and the implementation must be adjacent
    # in the source file, and overload order may matter:
```

(continues on next page)

(continued from previous page)

```
    # when two overloads may overlap, the more specific one
    # should come first.
    @overload
    def __getitem__(self, index: slice) -> Sequence[T]:
        pass  # Don't put code here

    # The implementation goes last, without @overload.
    # It may or may not have type hints; if it does,
    # these are checked against the overload definitions
    # as well as against the implementation body.
    def __getitem__(self, index: Union[int, slice]) -> Union[T, Sequence[T]]:
        # This is exactly the same as before.
        if isinstance(index, int):
            ...  # Return a T here
        elif isinstance(index, slice):
            ...  # Return a sequence of Ts here
        else:
            raise TypeError(...)
```

Calls to overloaded functions are type checked against the variants, not against the implementation. A call like `my_list[5]` would have type `T`, not `Union[T, Sequence[T]]` because it matches the first overloaded definition, and ignores the type annotations on the implementation of `__getitem__`. The code in the body of the definition of `__getitem__` is checked against the annotations on the the corresponding declaration. In this case the body is checked with `index: Union[int, slice]` and a return type `Union[T, Sequence[T]]`. If there are no annotations on the corresponding definition, then code in the function body is not type checked.

The annotations on the function body must be compatible with the types given for the overloaded variants listed above it. The type checker will verify that all the types listed the overloaded variants are compatible with the types given for the implementation. In this case it checks that the parameter type `int` and the return type `T` are compatible with `Union[int, slice]` and `Union[T, Sequence[T]]` for the first variant. For the second variant it verifies that the parameter type `slice` are the return type `Sequence[T]` are compatible with `Union[int, slice]` and `Union[T, Sequence[T]]`.

Overloaded function variants are still ordinary Python functions and they still define a single runtime object. There is no automatic dispatch happening, and you must manually handle the different types in the implementation (usually with `isinstance()` checks, as shown in the example).

The overload variants must be adjacent in the code. This makes code clearer, as you don't have to hunt for overload variants across the file.

Overloads in stub files are exactly the same, except there is no implementation.

Note: As generic type variables are erased at runtime when constructing instances of generic types, an overloaded function cannot have variants that only differ in a generic type argument, e.g. `List[int]` and `List[str]`.

Note: If you just need to constrain a type variable to certain types or subtypes, you can use a *value restriction*.

CHAPTER 13

Casts

Mypy supports type casts that are usually used to coerce a statically typed value to a subtype. Unlike languages such as Java or C#, however, mypy casts are only used as hints for the type checker, and they don't perform a runtime type check. Use the function `cast` to perform a cast:

```
from typing import cast, List

o = [1] # type: object
x = cast(List[int], o)  # OK
y = cast(List[str], o)  # OK (cast performs no actual runtime check)
```

To support runtime checking of casts such as the above, we'd have to check the types of all list items, which would be very inefficient for large lists. Use assertions if you want to perform an actual runtime check. Casts are used to silence spurious type checker warnings and give the type checker a little help when it can't quite understand what is going on.

You don't need a cast for expressions with type `Any`, or when assigning to a variable with type `Any`, as was explained earlier. You can also use `Any` as the cast target type – this lets you perform any operations on the result. For example:

```
from typing import cast, Any

x = 1
x + 'x'   # Type check error
y = cast(Any, x)
y + 'x'   # Type check OK (runtime error)
```

CHAPTER 14

Duck type compatibility

In Python, certain types are compatible even though they aren't subclasses of each other. For example, `int` objects are valid whenever `float` objects are expected. Mypy supports this idiom via *duck type compatibility*. This is supported for a small set of built-in types:

- `int` is duck type compatible with `float` and `complex`.
- `float` is duck type compatible with `complex`.
- In Python 2, `str` is duck type compatible with `unicode`.

For example, mypy considers an `int` object to be valid whenever a `float` object is expected. Thus code like this is nice and clean and also behaves as expected:

```
def degrees_to_radians(x: float) -> float:
    return math.pi * degrees / 180

n = 90  # Inferred type 'int'
print(degrees_to_radians(n))    # Okay!
```

You can also often use *Protocols and structural subtyping* to achieve a similar effect in a more principled and extensible fashion. Protocols don't apply to cases like `int` being compatible with `float`, since `float` is not a protocol class but a regular, concrete class, and many standard library functions expect concrete instances of `float` (or `int`).

Note: Note that in Python 2 a `str` object with non-ASCII characters is often *not valid* when a unicode string is expected. The mypy type system does not consider a string with non-ASCII values as a separate type so some programs with this kind of error will silently pass type checking. In Python 3 `str` and `bytes` are separate, unrelated types and this kind of error is easy to detect. This a good reason for preferring Python 3 over Python 2!

See *Text and AnyStr* for details on how to enforce that a value must be a unicode string in a cross-compatible way.

CHAPTER 15

Generics

15.1 Defining generic classes

The built-in collection classes are generic classes. Generic types have one or more type parameters, which can be arbitrary types. For example, `Dict[int, str]` has the type parameters `int` and `str`, and `List[int]` has a type parameter `int`.

Programs can also define new generic classes. Here is a very simple generic class that represents a stack:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class Stack(Generic[T]):
    def __init__(self) -> None:
        # Create an empty list with items of type T
        self.items = []  # type: List[T]

    def push(self, item: T) -> None:
        self.items.append(item)

    def pop(self) -> T:
        return self.items.pop()

    def empty(self) -> bool:
        return not self.items
```

The `Stack` class can be used to represent a stack of any type: `Stack[int]`, `Stack[Tuple[int, str]]`, etc.

Using `Stack` is similar to built-in container types:

```
# Construct an empty Stack[int] instance
stack = Stack[int]()
stack.push(2)
```

(continues on next page)

(continued from previous page)

```
stack.pop()
stack.push('x')         # Type error
```

Type inference works for user-defined generic types as well:

```
def process(stack: Stack[int]) -> None: ...

process(Stack())    # Argument has inferred type Stack[int]
```

Construction of instances of generic types is also type checked:

```
class Box(Generic[T]):
    def __init__(self, content: T) -> None:
        self.content = content

Box(1)  # OK, inferred type is Box[int]
Box[int](1)  # Also OK
s = 'some string'
Box[int](s)  # Type error
```

15.2 Generic class internals

You may wonder what happens at runtime when you index `Stack`. Actually, indexing `Stack` returns essentially a copy of `Stack` that returns instances of the original class on instantiation:

```
>>> print(Stack)
__main__.Stack
>>> print(Stack[int])
__main__.Stack[int]
>>> print(Stack[int]().__class__)
__main__.Stack
```

Note that built-in types `list`, `dict` and so on do not support indexing in Python. This is why we have the aliases `List`, `Dict` and so on in the `typing` module. Indexing these aliases gives you a class that directly inherits from the target class in Python:

```
>>> from typing import List
>>> List[int]
typing.List[int]
>>> List[int].__bases__
(<class 'list'>, typing.MutableSequence)
```

Generic types could be instantiated or subclassed as usual classes, but the above examples illustrate that type variables are erased at runtime. Generic `Stack` instances are just ordinary Python objects, and they have no extra runtime overhead or magic due to being generic, other than a metaclass that overloads the indexing operator.

15.3 Defining sub-classes of generic classes

User-defined generic classes and generic classes defined in `typing` can be used as base classes for another classes, both generic and non-generic. For example:

```
from typing import Generic, TypeVar, Mapping, Iterator, Dict

KT = TypeVar('KT')
VT = TypeVar('VT')

class MyMap(Mapping[KT, VT]]):  # This is a generic subclass of Mapping
    def __getitem__(self, k: KT) -> VT:
        ...  # Implementations omitted
    def __iter__(self) -> Iterator[KT]:
        ...
    def __len__(self) -> int:
        ...

items: MyMap[str, int]  # Okay

class StrDict(Dict[str, str]):  # This is a non-generic subclass of Dict
    def __str__(self) -> str:
        return 'StrDict({})'.format(super().__str__())

data: StrDict[int, int]  # Error! StrDict is not generic
data2: StrDict  # OK

class Receiver(Generic[T]):
    def accept(self, value: T) -> None:
        ...

class AdvancedReceiver(Receiver[T]):
    ...
```

Note: You have to add an explicit `Mapping` base class if you want mypy to consider a user-defined class as a mapping (and `Sequence` for sequences, etc.). This is because mypy doesn't use *structural subtyping* for these ABCs, unlike simpler protocols like `Iterable`, which use *structural subtyping*.

`Generic[...]` can be omitted from bases if there are other base classes that include type variables, such as `Mapping[KT, VT]` in the above example. If you include `Generic[...]` in bases, then it should list all type variables present in other bases (or more, if needed). The order of type variables is defined by the following rules:

- If `Generic[...]` is present, then the order of variables is always determined by their order in `Generic[...]`.
- If there are no `Generic[...]` in bases, then all type variables are collected in the lexicographic order (i.e. by first appearance).

For example:

```
from typing import Generic, TypeVar, Any

T = TypeVar('T')
S = TypeVar('S')
U = TypeVar('U')

class One(Generic[T]): ...
class Another(Generic[T]): ...

class First(One[T], Another[S]): ...
class Second(One[T], Another[S], Generic[S, U, T]): ...
```

(continues on next page)

(continued from previous page)

```
x: First[int, str]        # Here T is bound to int, S is bound to str
y: Second[int, str, Any]  # Here T is Any, S is int, and U is str
```

15.4 Generic functions

Generic type variables can also be used to define generic functions:

```
from typing import TypeVar, Sequence

T = TypeVar('T')      # Declare type variable

def first(seq: Sequence[T]) -> T:   # Generic function
    return seq[0]
```

As with generic classes, the type variable can be replaced with any type. That means `first` can be used with any sequence type, and the return type is derived from the sequence item type. For example:

```
# Assume first defined as above.

s = first('foo')      # s has type str.
n = first([1, 2, 3])  # n has type int.
```

Note also that a single definition of a type variable (such as `T` above) can be used in multiple generic functions or classes. In this example we use the same type variable in two generic functions:

```
from typing import TypeVar, Sequence

T = TypeVar('T')      # Declare type variable

def first(seq: Sequence[T]) -> T:
    return seq[0]

def last(seq: Sequence[T]) -> T:
    return seq[-1]
```

15.5 Generic methods and generic self

You can also define generic methods — just use a type variable in the method signature that is different from class type variables. In particular, `self` may also be generic, allowing a method to return the most precise type known at the point of access.

Note: This feature is experimental. Checking code with type annotations for self arguments is still not fully implemented. Mypy may disallow valid code or allow unsafe code.

In this way, for example, you can typecheck chaining of setter methods:

```
from typing import TypeVar
```

(continues on next page)

(continued from previous page)

```
T = TypeVar('T', bound='Shape')

class Shape:
    def set_scale(self: T, scale: float) -> T:
        self.scale = scale
        return self

class Circle(Shape):
    def set_radius(self, r: float) -> 'Circle':
        self.radius = r
        return self

class Square(Shape):
    def set_width(self, w: float) -> 'Square':
        self.width = w
        return self

circle = Circle().set_scale(0.5).set_radius(2.7)  # type: Circle
square = Square().set_scale(0.5).set_width(3.2)  # type: Square
```

Without using generic `self`, the last two lines could not be type-checked properly.

Other uses are factory methods, such as copy and deserialization. For class methods, you can also define generic `cls`, using `Type[T]`:

```
from typing import TypeVar, Tuple, Type

T = TypeVar('T', bound='Friend')

class Friend:
    other = None  # type: Friend

    @classmethod
    def make_pair(cls: Type[T]) -> Tuple[T, T]:
        a, b = cls(), cls()
        a.other = b
        b.other = a
        return a, b

class SuperFriend(Friend):
    pass

a, b = SuperFriend.make_pair()
```

Note that when overriding a method with generic `self`, you must either return a generic `self` too, or return an instance of the current class. In the latter case, you must implement this method in all future subclasses.

Note also that mypy cannot always verify that the implementation of a copy or a deserialization method returns the actual type of self. Therefore you may need to silence mypy inside these methods (but not at the call site), possibly by making use of the `Any` type.

15.6 Variance of generic types

There are three main kinds of generic types with respect to subtype relations between them: invariant, covariant, and contravariant. Assuming that we have a pair of types `A` and `B`, and `B` is a subtype of `A`, these are defined as follows:

- A generic class `MyCovGen[T, ...]` is called covariant in type variable `T` if `MyCovGen[B, ...]` is always a subtype of `MyCovGen[A, ...]`.
- A generic class `MyContraGen[T, ...]` is called contravariant in type variable `T` if `MyContraGen[A, ...]` is always a subtype of `MyContraGen[B, ...]`.
- A generic class `MyInvGen[T, ...]` is called invariant in `T` if neither of the above is true.

Let us illustrate this by few simple examples:

- `Union` is covariant in all variables: `Union[Cat, int]` is a subtype of `Union[Animal, int]`, `Union[Dog, int]` is also a subtype of `Union[Animal, int]`, etc. Most immutable containers such as `Sequence` and `FrozenSet` are also covariant.
- `Callable` is an example of type that behaves contravariant in types of arguments, namely `Callable[[Employee], int]` is a subtype of `Callable[[Manager], int]`. To understand this, consider a function:

```
def salaries(staff: List[Manager],
             accountant: Callable[[Manager], int]) -> List[int]: ...
```

  This function needs a callable that can calculate a salary for managers, and if we give it a callable that can calculate a salary for an arbitrary employee, it's still safe.

- `List` is an invariant generic type. Naively, one would think that it is covariant, but let us consider this code:

```
class Shape:
    pass

class Circle(Shape):
    def rotate(self):
        ...

def add_one(things: List[Shape]) -> None:
    things.append(Shape())

my_things: List[Circle] = []
add_one(my_things)      # This may appear safe, but...
my_things[0].rotate()   # ...this will fail
```

  Another example of invariant type is `Dict`. Most mutable containers are invariant.

By default, mypy assumes that all user-defined generics are invariant. To declare a given generic class as covariant or contravariant use type variables defined with special keyword arguments `covariant` or `contravariant`. For example:

```
from typing import Generic, TypeVar

T_co = TypeVar('T_co', covariant=True)

class Box(Generic[T_co]):  # this type is declared covariant
    def __init__(self, content: T_co) -> None:
        self._content = content

    def get_content(self) -> T_co:
        return self._content

def look_into(box: Box[Animal]): ...
```

(continues on next page)

(continued from previous page)

```
my_box = Box(Cat())
look_into(my_box)  # OK, but mypy would complain here for an invariant type
```

15.7 Type variables with value restriction

By default, a type variable can be replaced with any type. However, sometimes it's useful to have a type variable that can only have some specific types as its value. A typical example is a type variable that can only have values `str` and `bytes`:

```
from typing import TypeVar

AnyStr = TypeVar('AnyStr', str, bytes)
```

This is actually such a common type variable that `AnyStr` is defined in `typing` and we don't need to define it ourselves.

We can use `AnyStr` to define a function that can concatenate two strings or bytes objects, but it can't be called with other argument types:

```
from typing import AnyStr

def concat(x: AnyStr, y: AnyStr) -> AnyStr:
    return x + y

concat('a', 'b')    # Okay
concat(b'a', b'b')  # Okay
concat(1, 2)        # Error!
```

Note that this is different from a union type, since combinations of `str` and `bytes` are not accepted:

```
concat('string', b'bytes')   # Error!
```

In this case, this is exactly what we want, since it's not possible to concatenate a string and a bytes object! The type checker will reject this function:

```
def union_concat(x: Union[str, bytes], y: Union[str, bytes]) -> Union[str, bytes]:
    return x + y  # Error: can't concatenate str and bytes
```

Another interesting special case is calling `concat()` with a subtype of `str`:

```
class S(str): pass

ss = concat(S('foo'), S('bar'))
```

You may expect that the type of `ss` is `S`, but the type is actually `str`: a subtype gets promoted to one of the valid values for the type variable, which in this case is `str`. This is thus subtly different from *bounded quantification* in languages such as Java, where the return type would be `S`. The way mypy implements this is correct for `concat`, since `concat` actually returns a `str` instance in the above example:

```
>>> print(type(ss))
<class 'str'>
```

You can also use a `TypeVar` with a restricted set of possible values when defining a generic class. For example, mypy uses the type `typing.Pattern[AnyStr]` for the return value of `re.compile`, since regular expressions can be based on a string or a bytes pattern.

15.8 Type variables with upper bounds

A type variable can also be restricted to having values that are subtypes of a specific type. This type is called the upper bound of the type variable, and is specified with the `bound=...` keyword argument to `TypeVar`.

```
from typing import TypeVar, SupportsAbs

T = TypeVar('T', bound=SupportsAbs[float])
```

In the definition of a generic function that uses such a type variable `T`, the type represented by `T` is assumed to be a subtype of its upper bound, so the function can use methods of the upper bound on values of type `T`.

```
def largest_in_absolute_value(*xs: T) -> T:
    return max(xs, key=abs)  # Okay, because T is a subtype of SupportsAbs[float].
```

In a call to such a function, the type `T` must be replaced by a type that is a subtype of its upper bound. Continuing the example above,

```
largest_in_absolute_value(-3.5, 2)    # Okay, has type float.
largest_in_absolute_value(5+6j, 7)    # Okay, has type complex.
largest_in_absolute_value('a', 'b')   # Error: 'str' is not a subtype of␣
↪SupportsAbs[float].
```

Type parameters of generic classes may also have upper bounds, which restrict the valid values for the type parameter in the same way.

A type variable may not have both a value restriction (see *Type variables with value restriction*) and an upper bound.

15.9 Declaring decorators

One common application of type variable upper bounds is in declaring a decorator that preserves the signature of the function it decorates, regardless of that signature. Here's a complete example:

```
from typing import Any, Callable, TypeVar, Tuple, cast

FuncType = Callable[..., Any]
F = TypeVar('F', bound=FuncType)

# A decorator that preserves the signature.
def my_decorator(func: F) -> F:
    def wrapper(*args, **kwds):
        print("Calling", func)
        return func(*args, **kwds)
    return cast(F, wrapper)

# A decorated function.
@my_decorator
def foo(a: int) -> str:
    return str(a)
```

(continues on next page)

(continued from previous page)

```
# Another.
@my_decorator
def bar(x: float, y: float) -> Tuple[float, float, bool]:
    return (x, y, x > y)

a = foo(12)
reveal_type(a)  # str
b = bar(3.14, 0)
reveal_type(b)  # Tuple[float, float, bool]
foo('x')    # Type check error: incompatible type "str"; expected "int"
```

From the final block we see that the signatures of the decorated functions `foo()` and `bar()` are the same as those of the original functions (before the decorator is applied).

The bound on `F` is used so that calling the decorator on a non-function (e.g. `my_decorator(1)`) will be rejected.

Also note that the `wrapper()` function is not type-checked. Wrapper functions are typically small enough that this is not a big problem. This is also the reason for the `cast()` call in the `return` statement in `my_decorator()`. See *Casts*.

15.10 Generic protocols

Mypy supports generic protocols (see also *Protocols and structural subtyping*). Several *predefined protocols* are generic, such as `Iterable[T]`, and you can define additional generic protocols. Generic protocols mostly follow the normal rules for generic classes. Example:

```
from typing import TypeVar
from typing_extensions import Protocol

T = TypeVar('T')

class Box(Protocol[T]):
    content: T

def do_stuff(one: Box[str], other: Box[bytes]) -> None:
    ...

class StringWrapper:
    def __init__(self, content: str) -> None:
        self.content = content

class BytesWrapper:
    def __init__(self, content: bytes) -> None:
        self.content = content

do_stuff(StringWrapper('one'), BytesWrapper(b'other'))  # OK

x: Box[float] = ...
y: Box[int] = ...
x = y  # Error -- Box is invariant
```

The main difference between generic protocols and ordinary generic classes is that mypy checks that the declared variances of generic type variables in a protocol match how they are used in the protocol definition. The protocol in this example is rejected, since the type variable `T` is used covariantly as a return type, but the type variable is invariant:

```
from typing import TypeVar
from typing_extensions import Protocol

T = TypeVar('T')

class ReadOnlyBox(Protocol[T]):  # Error: covariant type variable expected
    def content(self) -> T: ...
```

This example correctly uses a covariant type variable:

```
from typing import TypeVar
from typing_extensions import Protocol

T_co = TypeVar('T_co', covariant=True)

class ReadOnlyBox(Protocol[T_co]):  # OK
    def content(self) -> T_co: ...

ax: ReadOnlyBox[float] = ...
ay: ReadOnlyBox[int] = ...
ax = ay  # OK -- ReadOnlyBox is covariant
```

See *Variance of generic types* for more about variance.

Generic protocols can also be recursive. Example:

```
T = TypeVar('T')

class Linked(Protocol[T]):
    val: T
    def next(self) -> 'Linked[T]': ...

class L:
    val: int

    ...  # details omitted

    def next(self) -> 'L':
        ...  # details omitted

def last(seq: Linked[T]) -> T:
    ...  # implementation omitted

result = last(L())  # Inferred type of 'result' is 'int'
```

CHAPTER 16

The mypy command line

This section documents many of mypy's command line flags. A quick summary of command line flags can always be printed using the `-h` flag (or its long form `--help`):

```
$ mypy -h
usage: mypy [-h] [-v] [-V] [--python-version x.y] [--platform PLATFORM] [-2]
            [--ignore-missing-imports]
            [--follow-imports {normal,silent,skip,error}]
            [--disallow-any-unimported] [--disallow-any-expr]
            [--disallow-any-decorated] [--disallow-any-explicit]
            [--disallow-any-generics] [--disallow-untyped-calls]
            [--disallow-untyped-defs] [--disallow-incomplete-defs]
            [--check-untyped-defs] [--disallow-subclassing-any]
            [--warn-incomplete-stub] [--disallow-untyped-decorators]
            [--warn-redundant-casts] [--no-warn-no-return] [--warn-return-any]
            [--warn-unused-ignores] [--warn-unused-configs]
            [--show-error-context] [--no-implicit-optional] [--no-incremental]
            [--quick-and-dirty] [--cache-dir DIR] [--cache-fine-grained]
            [--skip-version-check] [--no-strict-optional]
            [--strict-optional-whitelist [GLOB [GLOB ...]]]
            [--always-true NAME] [--always-false NAME] [--junit-xml JUNIT_XML]
            [--pdb] [--show-traceback] [--stats] [--inferstats]
            [--custom-typing MODULE] [--custom-typeshed-dir DIR]
            [--scripts-are-modules] [--config-file CONFIG_FILE]
            [--show-column-numbers] [--find-occurrences CLASS.MEMBER]
            [--strict] [--shadow-file SOURCE_FILE SHADOW_FILE]
            [--any-exprs-report DIR] [--cobertura-xml-report DIR]
            [--html-report DIR] [--linecount-report DIR]
            [--linecoverage-report DIR] [--memory-xml-report DIR]
            [--txt-report DIR] [--xml-report DIR] [--xslt-html-report DIR]
            [--xslt-txt-report DIR] [-m MODULE] [-p PACKAGE] [-c PROGRAM_TEXT]
            [files [files ...]]

(etc., too long to show everything here)
```

16.1 Specifying files and directories to be checked

You've already seen `mypy program.py` as a way to type check the file `program.py`. More generally you can pass any number of files and directories on the command line and they will all be type checked together.

- Files ending in `.py` (and stub files ending in `.pyi`) are checked as Python modules.
- Files not ending in `.py` or `.pyi` are assumed to be Python scripts and checked as such.
- Directories representing Python packages (i.e. containing a `__init__.py[i]` file) are checked as Python packages; all submodules and subpackages will be checked (subpackages must themselves have a `__init__.py[i]` file).
- Directories that don't represent Python packages (i.e. not directly containing an `__init__.py[i]` file) are checked as follows:
  - All `*.py[i]` files contained directly therein are checked as toplevel Python modules;
  - All packages contained directly therein (i.e. immediate subdirectories with an `__init__.py[i]` file) are checked as toplevel Python packages.

One more thing about checking modules and packages: if the directory *containing* a module or package specified on the command line has an `__init__.py[i]` file, mypy assigns these an absolute module name by crawling up the path until no `__init__.py[i]` file is found. For example, suppose we run the command `mypy foo/bar/baz.py` where `foo/bar/__init__.py` exists but `foo/__init__.py` does not. Then the module name assumed is `bar.baz` and the directory `foo` is added to mypy's module search path. On the other hand, if `foo/bar/__init__.py` did not exist, `foo/bar` would be added to the module search path instead, and the module name assumed is just `baz`.

If a script (a file not ending in `.py[i]`) is processed, the module name assumed is always `__main__` (matching the behavior of the Python interpreter).

16.2 Other ways of specifying code to be checked

The flag `-m` (long form: `--module`) lets you specify a module name to be found using the default module search path. The module name may contain dots. For example:

```
$ mypy -m html.parser
```

will type check the module `html.parser` (this happens to be a library stub).

The flag `-p` (long form: `--package`) is similar to `-m` but you give it a package name and it will type check all submodules and subpackages (recursively) of that package. (If you pass a package name to `-m` it will just type check the package's `__init__.py` and anything imported from there.) For example:

```
$ mypy -p html
```

will type check the entire `html` package (of library stubs). One can specify multiple packages and modules on the command line, for example:

```
$ mypy --package p.a --package p.b --module c
```

Finally the flag `-c` (long form: `--command`) will take a string from the command line and type check it as a small program. For example:

```
$ mypy -c 'x = [1, 2]; print(x())'
```

will type check that little program (and complain that `List[int]` is not callable).

16.3 Reading a list of files from a file

Finally, any command-line argument starting with `@` reads additional command-line arguments from the file following the `@` character. This is primarily useful if you have a file containing a list of files that you want to be type-checked: instead of using shell syntax like:

```
mypy $(cat file_of_files)
```

you can use this instead:

```
mypy @file_of_files
```

Such a file can also contain other flags, but a preferred way of reading flags (not files) from a file is to use a *configuration file*.

16.4 How imports are found

When mypy encounters an *import* statement it tries to find the module on the file system, similar to the way Python finds it. However, there are some differences.

First, mypy has its own search path. This is computed from the following items:

- The `MYPYPATH` environment variable (a colon-separated list of directories).
- The directories containing the sources given on the command line (see below).
- The relevant directories of the typeshed repo.

For sources given on the command line, the path is adjusted by crawling up from the given file or package to the nearest directory that does not contain an `__init__.py` or `__init__.pyi` file.

Second, mypy searches for stub files in addition to regular Python files and packages. The rules for searching a module `foo` are as follows:

- The search looks in each of the directories in the search path (see above) until a match is found.
- If a package named `foo` is found (i.e. a directory `foo` containing an `__init__.py` or `__init__.pyi` file) that's a match.
- If a stub file named `foo.pyi` is found, that's a match.
- If a Python module named `foo.py` is found, that's a match.

These matches are tried in order, so that if multiple matches are found in the same directory on the search path (e.g. a package and a Python file, or a stub file and a Python file) the first one in the above list wins.

In particular, if a Python file and a stub file are both present in the same directory on the search path, only the stub file is used. (However, if the files are in different directories, the one found in the earlier directory is used.)

NOTE: These rules are relevant to the following section too: the `--follow-imports` flag described below is applied *after* the above algorithm has determined which package, stub or module to use.

16.5 Following imports or not?

When you're first attacking a large existing codebase with mypy, you may only want to check selected files. For example, you may only want to check those files to which you have already added annotations. This is easily accomplished using a shell pipeline like this:

```
mypy $(find . -name \*.py | xargs grep -l '# type:')
```

(While there are many improvements possible to make this example more robust, this is not the place for a tutorial in shell programming.)

However, by default mypy doggedly tries to *follow imports*. This may cause several types of problems that you may want to silence during your initial conquest:

- Your code may import library modules for which no stub files exist yet. This can cause a lot of errors like the following:

  ```
  main.py:1: error: No library stub file for standard library module 'antigravity'
  main.py:2: error: No library stub file for module 'flask'
  main.py:3: error: Cannot find module named 'sir_not_appearing_in_this_film'
  ```

  If you see only a few of these you may be able to silence them by putting `# type: ignore` on the respective `import` statements, but it's usually easier to silence all such errors by using *–ignore-missing-imports*.

- Your project's directory structure may hinder mypy in finding certain modules that are part of your project, e.g. modules hidden away in a subdirectory that's not a package. You can usually deal with this by setting the `MYPYPATH` variable (see *How imports are found*).

- When following imports mypy may find a module that's part of your project but which you haven't annotated yet, mypy may report errors for the top level code in that module (where the top level includes class bodies and function/method default values). Here the `--follow-imports` flag comes in handy.

The `--follow-imports` flag takes a mandatory string value that can take one of four values. It only applies to modules for which a `.py` file is found (but no corresponding `.pyi` stub file) and that are not given on the command line. Passing a package or directory on the command line implies all modules in that package or directory. The four possible values are:

- `normal` (the default) follow imports normally and type check all top level code (as well as the bodies of all functions and methods with at least one type annotation in the signature).
- `silent` follow imports normally and even "type check" them normally, but *suppress any error messages*. This is typically the best option for a new codebase.
- `skip` *don't* follow imports, silently replacing the module (and everything imported *from* it) with an object of type `Any`. (This option used to be known as `--silent-imports` and while it is very powerful it can also cause hard-to-debug errors, hence the recommendation of using `silent` instead.)
- `error` the same behavior as `skip` but not quite as silent – it flags the import as an error, like this:

  ```
  main.py:1: note: Import of 'submodule' ignored
  main.py:1: note: (Using --follow-imports=error, module not passed on command line)
  ```

16.6 Disallow Any Flags

The `--disallow-any` family of flags disallows various types of `Any` in a module. The following options are available:

- `--disallow-any-unimported` disallows usage of types that come from unfollowed imports (such types become aliases for `Any`). Unfollowed imports occur either when the imported module does not exist or when `--follow-imports=skip` is set.
- `--disallow-any-expr` disallows all expressions in the module that have type `Any`. If an expression of type `Any` appears anywhere in the module mypy will output an error unless the expression is immediately used as an argument to `cast` or assigned to a variable with an explicit type annotation. In addition, declaring a

variable of type `Any` or casting to type `Any` is not allowed. Note that calling functions that take parameters of type `Any` is still allowed.

- `--disallow-any-decorated` disallows functions that have `Any` in their signature after decorator transformation.
- `--disallow-any-explicit` disallows explicit `Any` in type positions such as type annotations and generic type parameters.
- `--disallow-any-generics` disallows usage of generic types that do not specify explicit type parameters. Moreover, built-in collections (such as `list` and `dict`) become disallowed as you should use their aliases from the typing module (such as `List[int]` and `Dict[str, str]`).

16.7 Additional command line flags

Here are some more useful flags:

- `--ignore-missing-imports` suppresses error messages about imports that cannot be resolved (see *Following imports or not?* for some examples).
- `--no-strict-optional` disables strict checking of `Optional[...]` types and `None` values. With this option, mypy doesn't generally check the use of `None` values – they are valid everywhere. See *Disabling strict optional checking* for more about this feature.

  Note: Strict optional checking was enabled by default starting in mypy 0.600, and in previous versions it had to be explicitly enabled using `--strict-optional` (which is still accepted).
- `--disallow-untyped-defs` reports an error whenever it encounters a function definition without type annotations.
- `--check-untyped-defs` is less severe than the previous option – it type checks the body of every function, regardless of whether it has type annotations. (By default the bodies of functions without annotations are not type checked.) It will assume all arguments have type `Any` and always infer `Any` as the return type.
- `--disallow-incomplete-defs` reports an error whenever it encounters a partly annotated function definition.
- `--disallow-untyped-calls` reports an error whenever a function with type annotations calls a function defined without annotations.
- `--disallow-untyped-decorators` reports an error whenever a function with type annotations is decorated with a decorator without annotations.
- `--disallow-subclassing-any` reports an error whenever a class subclasses a value of type `Any`. This may occur when the base class is imported from a module that doesn't exist (when using *–ignore-missing-imports*) or is ignored due to *–follow-imports=skip* or a `# type: ignore` comment on the `import` statement. Since the module is silenced, the imported class is given a type of `Any`. By default mypy will assume that the subclass correctly inherited the base class even though that may not actually be the case. This flag makes mypy raise an error instead.
- Incremental mode enables a module cache, using results from previous runs to speed up type checking. Incremental mode can help when most parts of your program haven't changed since the previous mypy run.

  Incremental mode is the default and may be disabled with `--no-incremental`.
- `--cache-dir DIR` is a companion flag to incremental mode, which specifies where the cache files are written. By default this is `.mypy_cache` in the current directory. While the cache is only read in incremental mode, it is written even in non-incremental mode, in order to "warm" the cache. To disable writing the cache, use `--cache-dir=/dev/null` (UNIX) or `--cache-dir=nul` (Windows). Cache files belonging to a different mypy version are ignored. This flag can be useful for controlling cache use when using *remote caching*.

- `--quick-and-dirty` is an experimental, unsafe variant of *incremental mode*. Quick mode is faster than regular incremental mode, because it only re-checks modules that were modified since their cache file was last written (regular incremental mode also re-checks all modules that depend on one or more modules that were re-checked). Quick mode is unsafe because it may miss problems caused by a change in a dependency. Quick mode updates the cache, but regular incremental mode ignores cache files written by quick mode.
- `--python-executable EXECUTABLE` will have mypy collect type information from PEP 561 compliant packages installed for the Python executable `EXECUTABLE`. If not provided, mypy will use PEP 561 compliant packages installed for the Python executable running mypy. See *Using Installed Packages* for more on making PEP 561 compliant packages. This flag will attempt to set `--python-version` if not already set.
- `--python-version X.Y` will make mypy typecheck your code as if it were run under Python version X.Y. Without this option, mypy will default to using whatever version of Python is running mypy. Note that the `-2` and `--py2` flags are aliases for `--python-version 2.7`. See *Python version and system platform checks* for more about this feature. This flag will attempt to find a Python executable of the corresponding version to search for PEP 561 compliant packages. If you'd like to disable this, see `--no-site-packages` below.
- `--no-site-packages` will disable searching for PEP 561 compliant packages. This will also disable searching for a usable Python executable. Use this flag if mypy cannot find a Python executable for the version of Python being checked, and you don't need to use PEP 561 typed packages. Otherwise, use `--python-executable`.
- `--platform PLATFORM` will make mypy typecheck your code as if it were run under the the given operating system. Without this option, mypy will default to using whatever operating system you are currently using. See *Python version and system platform checks* for more about this feature.
- `--always-true NAME` will treat all variables named `NAME` as compile-time constants that are always true. May be repeated.
- `--always-false NAME` will treat all variables named `NAME` as compile-time constants that are always false. May be repeated.
- `--show-column-numbers` will add column offsets to error messages, for example, the following indicates an error in line 12, column 9 (note that column offsets are 0-based):

```
main.py:12:9: error: Unsupported operand types for / ("int" and "str")
```

- `--scripts-are-modules` will give command line arguments that appear to be scripts (i.e. files whose name does not end in `.py`) a module name derived from the script name rather than the fixed name `__main__`. This allows checking more than one script in a single mypy invocation. (The default `__main__` is technically more correct, but if you have many scripts that import a large package, the behavior enabled by this flag is often more convenient.)
- `--custom-typeshed-dir DIR` specifies the directory where mypy looks for typeshed stubs, instead of the typeshed that ships with mypy. This is primarily intended to make it easier to test typeshed changes before submitting them upstream, but also allows you to use a forked version of typeshed.
- `--config-file CONFIG_FILE` causes configuration settings to be read from the given file. By default settings are read from `mypy.ini` or `setup.cfg` in the current directory, or `.mypy.ini` in the user home directory. Settings override mypy's built-in defaults and command line flags can override settings. See *The mypy configuration file* for the syntax of configuration files.
- `--junit-xml JUNIT_XML` will make mypy generate a JUnit XML test result document with type checking results. This can make it easier to integrate mypy with continuous integration (CI) tools.
- `--find-occurrences CLASS.MEMBER` will make mypy print out all usages of a class member based on static type information. This feature is experimental.
- `--cobertura-xml-report DIR` causes mypy to generate a Cobertura XML type checking coverage report.

- `--warn-no-return` causes mypy to generate errors for missing return statements on some execution paths. Mypy doesn't generate these errors for functions with `None` or `Any` return types. Mypy also currently ignores functions with an empty body or a body that is just ellipsis (`...`), since these can be valid as abstract methods. This option is on by default.
- `--warn-return-any` causes mypy to generate a warning when returning a value with type `Any` from a function declared with a non- `Any` return type.
- `--strict` mode enables all optional error checking flags. You can see the list of flags enabled by strict mode in the full `mypy -h` output.
- `--shadow-file SOURCE_FILE SHADOW_FILE`: when mypy is asked to typecheck `SOURCE_FILE`, this makes it read from and typecheck the contents of `SHADOW_FILE` instead. However, diagnostics will continue to refer to `SOURCE_FILE`. Specifying this argument multiple times (`--shadow-file X1 Y1 --shadow-file X2 Y2`) will allow mypy to perform multiple substitutions.

  This allows tooling to create temporary files with helpful modifications without having to change the source file in place. For example, suppose we have a pipeline that adds `reveal_type` for certain variables. This pipeline is run on `original.py` to produce `temp.py`. Running `mypy --shadow-file original.py temp.py original.py` will then cause mypy to typecheck the contents of `temp.py` instead of `original.py`, but error messages will still reference `original.py`.
- `--no-implicit-optional` causes mypy to stop treating arguments with a `None` default value as having an implicit `Optional[...]` type.

For the remaining flags you can read the full `mypy -h` output.

Note: Command line flags are liable to change between releases.

16.8 Integrating mypy into another Python application

It is possible to integrate mypy into another Python 3 application by importing `mypy.api` and calling the `run` function with a parameter of type `List[str]`, containing what normally would have been the command line arguments to mypy.

Function `run` returns a `Tuple[str, str, int]`, namely `(<normal_report>, <error_report>, <exit_status>)`, in which `<normal_report>` is what mypy normally writes to `sys.stdout`, `<error_report>` is what mypy normally writes to `sys.stderr` and `exit_status` is the exit status mypy normally returns to the operating system.

A trivial example of using the api is the following:

```
import sys
from mypy import api

result = api.run(sys.argv[1:])

if result[0]:
    print('\nType checking report:\n')
    print(result[0])  # stdout

if result[1]:
    print('\nError report:\n')
    print(result[1])  # stderr

print ('\nExit status:', result[2])
```

CHAPTER 17

The mypy configuration file

Mypy supports reading configuration settings from a file. By default it uses the file `mypy.ini` with fallback to `setup.cfg` in the current directory, or `.mypy.ini` in the user home directory if none of them are found; the `--config-file` command-line flag can be used to read a different file instead (see *–config-file*).

It is important to understand that there is no merging of configuration files, as it would lead to ambiguity. The `--config-file` flag has the highest precedence and must be correct; otherwise mypy will report an error and exit. Without command line option, mypy will look for defaults, but will use only one of them. The first one to read is `mypy.ini`, and then `setup.cfg`.

Most flags correspond closely to *command-line flags* but there are some differences in flag names and some flags may take a different value based on the module being processed.

The configuration file format is the usual ini file format. It should contain section names in square brackets and flag settings of the form *NAME* = *VALUE*. Comments start with # characters.

- A section named `[mypy]` must be present. This specifies the global flags. The `setup.cfg` file is an exception to this.
- Additional sections named `[mypy-PATTERN1,PATTERN2,...]` may be present, where `PATTERN1`, `PATTERN2`, etc., are comma-separated patterns of the form `dotted_module_name` or `dotted_module_name.*`. These sections specify additional flags that only apply to *modules* whose name matches at least one of the patterns.

  A pattern of the form `dotted_module_name` matches only the named module, while `dotted_module_name.*` matches `dotted_module_name` and any submodules (so `foo.bar.*` would match all of `foo.bar`, `foo.bar.baz`, and `foo.bar.baz.quux`).

Note: The `warn_unused_configs` flag may be useful to debug misspelled section names.

17.1 Global flags

The following global flags may only be set in the global section (`[mypy]`).

- `python_version` (string) specifies the Python version used to parse and check the target program. The format is `DIGIT.DIGIT` for example `2.7`. The default is the version of the Python interpreter used to run mypy.
- `platform` (string) specifies the OS platform for the target program, for example `darwin` or `win32` (meaning OS X or Windows, respectively). The default is the current platform as revealed by Python's `sys.platform` variable.
- `always_true` (comma-separated list of strings) gives variable names that will be treated as compile-time constants that are always true.
- `always_false` (comma-separated list of strings) gives variable names that will be treated as compile-time constants that are always false.
- `custom_typing_module` (string) specifies the name of an alternative module which is to be considered equivalent to the `typing` module.
- `custom_typeshed_dir` (string) specifies the name of an alternative directory which is used to look for stubs instead of the default `typeshed` directory.
- `mypy_path` (string) specifies the paths to use, after trying the paths from `MYPYPATH` environment variable. Useful if you'd like to keep stubs in your repo, along with the config file.
- `warn_incomplete_stub` (Boolean, default False) warns for missing type annotation in typeshed. This is only relevant in combination with `check_untyped_defs`.
- `warn_redundant_casts` (Boolean, default False) warns about casting an expression to its inferred type.
- `warn_unused_configs` (Boolean, default False) warns about per-module sections in the config file that didn't match any files processed in the current run.
- `scripts_are_modules` (Boolean, default False) makes script `x` become module `x` instead of `__main__`. This is useful when checking multiple scripts in a single run.
- `verbosity` (integer, default 0) controls how much debug output will be generated. Higher numbers are more verbose.
- `pdb` (Boolean, default False) invokes pdb on fatal error.
- `show_traceback` (Boolean, default False) shows traceback on fatal error.
- `dump_type_stats` (Boolean, default False) dumps stats about type definitions.
- `dump_inference_stats` (Boolean, default False) dumps stats about type inference.
- `incremental` (Boolean, default True) enables *incremental mode*.
- `cache_dir` (string, default `.mypy_cache`) stores module cache info in the given folder in *incremental mode*. The cache is only read in incremental mode, but it is always written unless the value is set to `/dev/null` (UNIX) or `nul` (Windows).
- `quick_and_dirty` (Boolean, default False) enables *quick mode*.
- `show_error_context` (Boolean, default False) shows context notes before errors.
- `show_column_numbers` (Boolean, default False) shows column numbers in error messages.

17.2 Per-module flags

The following flags may vary per module. They may also be specified in the global section; the global section provides defaults which are overridden by the pattern sections matching the module name.

Note: If multiple pattern sections match a module, the options from the most specific section are used where they disagree. This means that `foo.bar` will take values from sections with the patterns `foo.bar`, `foo.bar.*`, and `foo.*`, but when they specify different values, it will use values from `foo.bar` before `foo.bar.*` before `foo.*`.

- `follow_imports` (string, default `normal`) directs what to do with imports when the imported module is found as a `.py` file and not part of the files, modules and packages on the command line. The four possible values are `normal`, `silent`, `skip` and `error`. For explanations see the discussion for the *–follow-imports* command line flag. Note that if pattern matching is used, the pattern should match the name of the *imported* module, not the module containing the import statement.
- `follow_imports_for_stubs` (Boolean, default false) determines whether to respect the `follow_imports` setting even for stub (`.pyi`) files. Used in conjunction with `follow_imports=skip`, this can be used to suppress the import of a module from `typeshed`, replacing it with *Any*. Used in conjuncation with `follow_imports=error`, this can be used to make any use of a particular `typeshed` module an error.
- `ignore_missing_imports` (Boolean, default False) suppress error messages about imports that cannot be resolved. Note that if pattern matching is used, the pattern should match the name of the *imported* module, not the module containing the import statement.
- `silent_imports` (Boolean, deprecated) equivalent to `follow_imports=skip` plus `ignore_missing_imports=True`.
- `almost_silent` (Boolean, deprecated) equivalent to `follow_imports=skip`.
- `strict_optional` (Boolean, default True) enables or disables strict Optional checks. If False, mypy treats `None` as compatible with every type.

  Note:: This was False by default in mypy versions earlier than 0.600.
- `disallow_any_unimported` (Boolean, default false) disallows usage of types that come from unfollowed imports (such types become aliases for `Any`).
- `disallow_any_expr` (Boolean, default false) disallows all expressions in the module that have type `Any`.
- `disallow_any_decorated` (Boolean, default false) disallows functions that have `Any` in their signature after decorator transformation.
- `disallow_any_explicit` (Boolean, default false) disallows explicit `Any` in type positions such as type annotations and generic type parameters.
- `disallow_any_generics` (Boolean, default false) disallows usage of generic types that do not specify explicit type parameters.
- `disallow_subclassing_any` (Boolean, default False) disallows subclassing a value of type `Any`. See *–disallow-subclassing-any* option.
- `disallow_untyped_calls` (Boolean, default False) disallows calling functions without type annotations from functions with type annotations.
- `disallow_untyped_defs` (Boolean, default False) disallows defining functions without type annotations or with incomplete type annotations.
- `check_untyped_defs` (Boolean, default False) type-checks the interior of functions without type annotations.
- `debug_cache` (Boolean, default False) writes the incremental cache JSON files using a more readable, but slower format.
- `show_none_errors` (Boolean, default True) shows errors related to strict `None` checking, if the global `strict_optional` flag is enabled.

- `ignore_errors` (Boolean, default False) ignores all non-fatal errors.
- `warn_no_return` (Boolean, default True) shows errors for missing return statements on some execution paths.
- `warn_return_any` (Boolean, default False) shows a warning when returning a value with type `Any` from a function declared with a non- `Any` return type.
- `warn_unused_ignores` (Boolean, default False) warns about unneeded `# type: ignore` comments.
- `strict_boolean` (Boolean, default False) makes using non-boolean expressions in conditions an error.
- `no_implicit_optional` (Boolean, default false) changes the treatment of arguments with a default value of None by not implicitly making their type Optional

17.3 Examples

You might put this in your `mypy.ini` file at the root of your repo:

```
[mypy]
python_version = 2.7
[mypy-foo.*]
disallow_untyped_defs = True
```

This automatically sets `--python-version 2.7` (a.k.a. `--py2`) for all mypy runs in this tree, and also selectively turns on the `--disallow-untyped-defs` flag for all modules in the `foo` package. This issues an error for function definitions without type annotations in that subdirectory only.

If you would like to ignore specific imports, instead of ignoring all missing imports with `--ignore-missing-imports`, use a section of the configuration file per module such as the following to ignore missing imports from `lib_module`:

```
[mypy-lib_module]
ignore_missing_imports = True
```

Note: Configuration flags are liable to change between releases.

CHAPTER 18

Mypy daemon (mypy server)

Instead of running mypy as a command-line tool, you can also run it as a long-running daemon (server) process and use a command-line client to send type-checking requests to the server. This way mypy can perform type checking much faster, since program state cached from previous runs is kept in memory and doesn't have to be read from the file system on each run. The server also uses finer-grained dependency tracking to reduce the amount of work that needs to be done.

If you have a large codebase to check, running mypy using the mypy daemon can be *10 or more times faster* than the regular command-line `mypy` tool, especially if your workflow involves running mypy repeatedly after small edits – which is often a good idea, as this way you'll find errors sooner.

Note: The mypy daemon is experimental. In particular, the command-line interface may change in future mypy releases.

Note: The mypy daemon currently supports macOS and Linux only.

Note: Each mypy daemon process supports one user and one set of source files, and it can only process one type checking request at a time. You can run multiple mypy daemon processes to type check multiple repositories.

18.1 Basic usage

The client utility `dmypy` is used to control the mypy daemon. Use `dmypy start -- <flags>` to start the daemon. You can use almost arbitrary mypy flags after `--`. The daemon will always run on the current host. Example:

```
dmypy start -- --follow-imports=skip
```

Note: You'll need to use either the `--follow-imports=skip` or the `--follow-imports=error` option with dmypy because the current implementation can't follow imports. See *Following imports or not?* for details on how these work. You can also define these using a *configuration file*.

The daemon will not type check anything when it's started. Use `dmypy check <files>` to check some files (or directories):

```
dmypy check prog.py pkg1/ pkg2/
```

You need to provide all files or directories you want to type check (other than stubs) as arguments. This is a result of the `--follow-imports` restriction mentioned above.

The initial run will process all the code and may take a while to finish, but subsequent runs will be quick, especially if you've only changed a few files. You can use *remote caching* to speed up the initial run. The speedup can be significant if you have a large codebase.

18.2 Additional features

You have precise control over the lifetime of the daemon process:

- `dymypy stop` stops the daemon.
- `dmypy restart -- <flags>` restarts the daemon. The flags are the same as with `dmypy start`. This is equivalent to a stop command followed by a start.
- Use `dmypy start --timeout SECONDS -- <flags>` (or `dmypy restart --timeout SECONDS -- <flags>`) to automatically shut down the daemon after inactivity. By default, the daemon runs until it's explicitly stopped.

Use `dmypy --help` for help on additional commands and command-line options not discussed here, and `dmypy <command> --help` for help on command-specific options.

18.3 Limitations

- You have to use either the `--follow-imports=skip` or the `--follow-imports=error` option because of an implementation limitation. This can be defined through the command line or through a *configuration file*.
- Windows is not supported.

CHAPTER 19

Supported Python features and modules

A list of unsupported Python features is maintained in the mypy wiki:

- Unsupported Python features

19.1 Runtime definition of methods and functions

By default, mypy will complain if you add a function to a class or module outside its definition – but only if this is visible to the type checker. This only affects static checking, as mypy performs no additional type checking at runtime. You can easily work around this. For example, you can use dynamically typed code or values with `Any` types, or you can use `setattr` or other introspection features. However, you need to be careful if you decide to do this. If used indiscriminately, you may have difficulty using static typing effectively, since the type checker cannot see functions defined at runtime.

CHAPTER 20

New features in Python 3.6

Python 3.6 was released in December 2016. As of mypy 0.510 all language features new in Python 3.6 are supported.

20.1 Syntax for variable annotations (PEP 526)

Python 3.6 feature: variables (in global, class or local scope) can now have type annotations using either of the two forms:

```
from typing import Optional
foo: Optional[int]
bar: List[str] = []
```

Mypy fully supports this syntax, interpreting them as equivalent to

```
foo = None  # type: Optional[int]
bar = []  # type: List[str]
```

An additional feature defined in PEP 526 is also supported: you can mark names intended to be used as class variables with `ClassVar`. In a pinch you can also use ClassVar in `# type` comments. Example:

```
from typing import ClassVar

class C:
    x: int  # instance variable
    y: ClassVar[int]  # class variable
    z = None  # type: ClassVar[int]

    def foo(self) -> None:
        self.x = 0  # OK
        self.y = 0  # Error: Cannot assign to class variable "y" via instance

C.y = 0  # This is OK
```

20.2 Literal string formatting (PEP 498)

Python 3.6 feature: string literals of the form `f"text {expression} text"` evaluate `expression` using the current evaluation context (locals and globals).

Mypy fully supports this syntax and type-checks the `expression`.

20.3 Underscores in numeric literals (PEP 515)

Python 3.6 feature: numeric literals can contain underscores, e.g. `1_000_000`.

Mypy fully supports this syntax:

```
precise_val = 1_000_000.000_000_1
hexes: List[int] = []
hexes.append(0x_FF_FF_FF_FF)
```

20.4 Asynchronous generators (PEP 525) and comprehensions (PEP 530)

Python 3.6 allows coroutines defined with `async def` (PEP 492) to be generators, i.e. contain `yield` expressions, and introduces a syntax for asynchronous comprehensions. Mypy fully supports these features, for example:

```
from typing import AsyncIterator

async def gen() -> AsyncIterator[bytes]:
    lst = [b async for b in gen()]  # Inferred type is "List[bytes]"
    yield 'no way'  # Error: Incompatible types (got "str", expected "bytes")
```

20.5 New named tuple syntax

Python 3.6 supports an alternative syntax for named tuples. See *Named tuples*.

CHAPTER 21

Additional features

This section discusses various features outside core mypy features.

21.1 The attrs package

attrs is a package that lets you define classes without writing boilerplate code. Mypy can detect uses of the package and will generate the necessary method definitions for decorated classes using the type annotations it finds. Type annotations can be added as follows:

```
import attr
@attr.s
class A:
    one: int = attr.ib()          # Variable annotation (Python 3.6+)
    two = attr.ib()  # type: int  # Type comment
    three = attr.ib(type=int)     # type= argument
```

If you're using `auto_attribs=True` you must use variable annotations.

```
import attr
@attr.s(auto_attribs=True)
class A:
    one: int
    two: int = 7
    three: int = attr.ib(8)
```

Typeshed has a couple of "white lie" annotations to make type checking easier. `attr.ib` and `attr.Factory` actually return objects, but the annotation says these return the types that they expect to be assigned to. That enables this to work:

```
import attr
from typing import Dict
@attr.s(auto_attribs=True)
class A:
```

(continues on next page)

(continued from previous page)

```
    one: int = attr.ib(8)
    two: Dict[str, str] = attr.Factory(dict)
    bad: str = attr.ib(16)    # Error: can't assign int to str
```

21.1.1 Caveats/Known Issues

- The detection of attr classes and attributes works by function name only. This means that if you have your own helper functions that, for example, `return attr.ib()` mypy will not see them.
- All boolean arguments that mypy cares about must be literal `True` or `False`. e.g the following will not work:

```
import attr
YES = True
@attr.s(init=YES)
class A:
    ...
```

- Currently, `converter` only supports named functions. If mypy finds something else it will complain about not understanding the argument and the type annotation in `__init__` will be replaced by `Any`.
- Validator decorators and default decorators are not type-checked against the attribute they are setting/validating.
- Method definitions added by mypy currently overwrite any existing method definitions.

21.2 Using a remote cache to speed up mypy runs

Mypy performs type checking *incrementally*, reusing results from previous runs to speed up successive runs. If you are type checking a large codebase, mypy can still be sometimes slower than desirable. For example, if you create a new branch based on a much more recent commit than the target of the previous mypy run, mypy may have to process almost every file, as a large fraction of source files may have changed. This can also happen after you've rebased a local branch.

Mypy supports using a *remote cache* to improve performance in cases such as the above. In a large codebase, remote caching can sometimes speed up mypy runs by a factor of 10, or more.

Mypy doesn't include all components needed to set this up – generally you will have to perform some simple integration with your Continuous Integration (CI) or build system to configure mypy to use a remote cache. This discussion assumes you have a CI system set up for the mypy build you want to speed up, and that you are using a central git repository. Generalizing to different environments should not be difficult.

Here are the main components needed:

- A shared repository for storing mypy cache files for all landed commits.
- CI build that uploads mypy incremental cache files to the shared repository for each commit for which the CI build runs.
- A wrapper script around mypy that developers use to run mypy with remote caching enabled.

Below we discuss each of these components in some detail.

21.2.1 Shared repository for cache files

You need a repository that allows you to upload mypy cache files from your CI build and make the cache files available for download based on a commit id. A simple approach would be to produce an archive of the `.mypy_cache`

directory (which contains the mypy cache data) as a downloadable *build artifact* from your CI build (depending on the capabilities of your CI system). Alternatively, you could upload the data to a web server or to S3, for example.

21.2.2 Continuous Integration build

The CI build would run a regular mypy build and create an archive containing the `.mypy_cache` directory produced by the build. Finally, it will produce the cache as a build artifact or upload it to a repository where it is accessible by the mypy wrapper script.

Your CI script might work like this:

- Run mypy normally. This will generate cache data under the `.mypy_cache` directory.
- Create a tarball from the `.mypy_cache` directory.
- Determine the current git master branch commit id (say, using `git rev-parse HEAD`).
- Upload the tarball to the shared repository with a name derived from the commit id.

21.2.3 Mypy wrapper script

The wrapper script is used by developers to run mypy locally during development instead of invoking mypy directly. The wrapper first populates the local `.mypy_cache` directory from the shared repository and then runs a normal incremental build.

The wrapper script needs some logic to determine the most recent central repository commit (by convention, the `origin/master` branch for git) the local development branch is based on. In a typical git setup you can do it like this:

```
git merge-base HEAD origin/master
```

The next step is to download the cache data (contents of the `.mypy_cache` directory) from the shared repository based on the commit id of the merge base produced by the git command above. The script will decompress the data so that mypy will start with a fresh `.mypy_cache`. Finally, the script runs mypy normally. And that's all!

21.2.4 Caching with mypy daemon

You can also use remote caching with the *mypy daemon*. The remote cache will significantly speed up the the first `dmypy check` run after starting or restarting the daemon.

The mypy daemon requires extra fine-grained dependency data in the cache files which aren't included by default. To use caching with the mypy daemon, use the `--cache-fine-grained` option in your CI build:

```
$ mypy --cache-fine-grained <args...>
```

This flag adds extra information for the daemon to the cache. In order to use this extra information, you will also need to use the `--use-fine-grained-cache` option with `dymypy start` or `dmypy restart`. Example:

```
$ dmypy start -- --use-fine-grained-cache <options...>
```

Now your first `dmypy check` run should be much faster, as it can use cache information to avoid processing the whole program.

21.2.5 Refinements

There are several optional refinements that may improve things further, at least if your codebase is hundreds of thousands of lines or more:

- If the wrapper script determines that the merge base hasn't changed from a previous run, there's no need to download the cache data and it's better to instead reuse the existing local cache data.
- If you use the mypy daemon, you may want to restart the daemon each time after the merge base or local branch has changed to avoid processing a potentially large number of changes in an incremental build, as this can be much slower than downloading cache data and restarting the daemon.
- If the current local branch is based on a very recent master commit, the remote cache data may not yet be available for that commit, as there will necessarily be some latency to build the cache files. It may be a good idea to look for cache data for, say, the 5 latest master commits and use the most recent data that is available.
- If the remote cache is not accessible for some reason (say, from a public network), the script can still fall back to a normal incremental build.
- You can have multiple local cache directories for different local branches using the `--cache-dir` option. If the user switches to an existing branch where downloaded cache data is already available, you can continue to use the existing cache data instead of redownloading the data.
- You can set up your CI build to use a remote cache to speed up the CI build. This would be particularly useful if each CI build starts from a fresh state without access to cache files from previous builds. It's still recommended to run a full, non-incremental mypy build to create the cache data, as repeatedly updating cache data incrementally could result in drift over a long time period (due to a mypy caching issue, perhaps).

CHAPTER 22

Using Installed Packages

PEP 561 specifies how to mark a package as supporting type checking. Below is a summary of how to create PEP 561 compatible packages and have mypy use them in type checking.

22.1 Using PEP 561 compatible packages with mypy

Generally, you do not need to do anything to use installed packages that support typing for the Python executable used to run mypy. Note that most packages do not support typing. Packages that do support typing should be automatically picked up by mypy and used for type checking.

By default, mypy searches for packages installed for the Python executable running mypy. It is highly unlikely you want this situation if you have installed typed packages in another Python's package directory.

Generally, you can use the `--python-version` flag and mypy will try to find the correct package directory. If that fails, you can use the `--python-executable` flag to point to the exact executable, and mypy will find packages installed for that Python executable.

Note that mypy does not support some more advanced import features, such as zip imports, namespace packages, and custom import hooks.

If you do not want to use typed packages, use the `--no-site-packages` flag to disable searching.

22.2 Making PEP 561 compatible packages

PEP 561 notes three main ways to distribute type information. The first is a package that has only inline type annotations in the code itself. The second is a package that ships stub files with type information alongside the runtime code. The third method, also known as a "stub only package" is a package that ships type information for a package separately as stub files.

If you would like to publish a library package to a package repository (e.g. PyPI) for either internal or external use in type checking, packages that supply type information via type comments or annotations in the code should put a `py.typed` in their package directory. For example, with a directory structure as follows:

```
setup.py
package_a/
    __init__.py
    lib.py
    py.typed
```

the setup.py might look like:

```
from distutils.core import setup

setup(
    name="SuperPackageA",
    author="Me",
    version="0.1",
    package_data={"package_a": ["py.typed"]},
    packages=["package_a"]
)
```

Some packages have a mix of stub files and runtime files. These packages also require a `py.typed` file. An example can be seen below:

```
setup.py
package_b/
    __init__.py
    lib.py
    lib.pyi
    py.typed
```

the setup.py might look like:

```
from distutils.core import setup

setup(
    name="SuperPackageB",
    author="Me",
    version="0.1",
    package_data={"package_b": ["py.typed", "lib.pyi"]},
    packages=["package_b"]
)
```

In this example, both `lib.py` and `lib.pyi` exist. At runtime, the Python interpeter will use `lib.py`, but mypy will use `lib.pyi` instead.

If the package is stub-only (not imported at runtime), the package should have a prefix of the runtime package name and a suffix of `-stubs`. A `py.typed` file is not needed for stub-only packages. For example, if we had stubs for `package_c`, we might do the following:

```
setup.py
package_c-stubs/
    __init__.pyi
    lib.pyi
```

the setup.py might look like:

```
from distutils.core import setup

setup(
```

(continues on next page)

(continued from previous page)

```
    name="SuperPackageC",
    author="Me",
    version="0.1",
    package_data={"package_c-stubs": ["__init__.pyi", "lib.pyi"]},
    packages=["package_c-stubs"]
)
```

CHAPTER 23

Common issues

This section has examples of cases when you need to update your code to use static typing, and ideas for working around issues if mypy doesn't work as expected. Statically typed code is often identical to normal Python code, but sometimes you need to do things slightly differently.

23.1 Can't install mypy using pip

If installation fails, you've probably hit one of these issues:

- Mypy needs Python 3.4 or later to run.
- You may have to run pip like this: `python3 -m pip install mypy`.

23.2 No errors reported for obviously wrong code

There are several common reasons why obviously wrong code is not flagged as an error.

- **The function containing the error is not annotated.** Functions that do not have any annotations (neither for any argument nor for the return type) are not type-checked, and even the most blatant type errors (e.g. `2 + 'a'`) pass silently. The solution is to add annotations. Where that isn't possible, functions without annotations can be checked using `--check-untyped-defs`.

  Example:

  ```
  def foo(a):
      return '(' + a.split() + ')'  # No error!
  ```

  This gives no error even though `a.split()` is "obviously" a list (the author probably meant `a.strip()`). The error is reported once you add annotations:

```
def foo(a: str) -> str:
    return '(' + a.split() + ')'
# error: Unsupported operand types for + ("str" and List[str])
```

If you don't know what types to add, you can use `Any`, but beware:

- **One of the values involved has type 'Any'.** Extending the above example, if we were to leave out the annotation for `a`, we'd get no error:

```
def foo(a) -> str:
    return '(' + a.split() + ')'  # No error!
```

The reason is that if the type of `a` is unknown, the type of `a.split()` is also unknown, so it is inferred as having type `Any`, and it is no error to add a string to an `Any`.

If you're having trouble debugging such situations, *reveal_type()* might come in handy.

Note that sometimes library stubs have imprecise type information, e.g. the `pow()` builtin returns `Any` (see typeshed issue 285 for the reason).

- **Some imports may be silently ignored**. Another source of unexpected `Any` values are the *"–ignore-missing-imports"* and *"–follow-imports=skip"* flags. When you use `--ignore-missing-imports`, any imported module that cannot be found is silently replaced with `Any`. When using `--follow-imports=skip` the same is true for modules for which a `.py` file is found but that are not specified on the command line. (If a `.pyi` stub is found it is always processed normally, regardless of the value of `--follow-imports`.) To help debug the former situation (no module found at all) leave out `--ignore-missing-imports`; to get clarity about the latter use `--follow-imports=error`. You can read up about these and other useful flags in *The mypy command line*.

- **A function annotated as returning a non-optional type returns 'None' and mypy doesn't complain**.

```
def foo() -> str:
    return None  # No error!
```

You may have disabled strict optional checking (see *Disabling strict optional checking* for more).

23.3 Spurious errors and locally silencing the checker

You can use a `# type: ignore` comment to silence the type checker on a particular line. For example, let's say our code is using the C extension module `frobnicate`, and there's no stub available. Mypy will complain about this, as it has no information about the module:

```
import frobnicate  # Error: No module "frobnicate"
frobnicate.start()
```

You can add a `# type: ignore` comment to tell mypy to ignore this error:

```
import frobnicate  # type: ignore
frobnicate.start()  # Okay!
```

The second line is now fine, since the ignore comment causes the name `frobnicate` to get an implicit `Any` type.

Note: The `# type: ignore` comment will only assign the implicit `Any` type if mypy cannot find information about that particular module. So, if we did have a stub available for `frobnicate` then mypy would ignore the `# type: ignore` comment and typecheck the stub as usual.

23.4 Unexpected errors about 'None' and/or 'Optional' types

Starting from mypy 0.600, mypy uses *strict optional checking* by default, and `None` is not compatible with non-optional types. It's easy to switch back to the older behavior where `None` was compatible with arbitrary types (see *Disabling strict optional checking*).

23.5 Types of empty collections

You often need to specify the type when you assign an empty list or dict to a new variable, as mentioned earlier:

```
a = []  # type: List[int]
```

Without the annotation mypy can't always figure out the precise type of `a`.

You can use a simple empty list literal in a dynamically typed function (as the type of `a` would be implicitly `Any` and need not be inferred), if type of the variable has been declared or inferred before, or if you perform a simple modification operation in the same scope (such as `append` for a list):

```
a = []  # Okay because followed by append, inferred type List[int]
for i in range(n):
    a.append(i * i)
```

However, in more complex cases an explicit type annotation can be required (mypy will tell you this). Often the annotation can make your code easier to understand, so it doesn't only help mypy but everybody who is reading the code!

23.6 Redefinitions with incompatible types

Each name within a function only has a single 'declared' type. You can reuse for loop indices etc., but if you want to use a variable with multiple types within a single function, you may need to declare it with the `Any` type.

```
def f() -> None:
    n = 1
    ...
    n = 'x'         # Type error: n has type int
```

Note: This limitation could be lifted in a future mypy release.

Note that you can redefine a variable with a more *precise* or a more concrete type. For example, you can redefine a sequence (which does not support `sort()`) as a list and sort it in-place:

```
def f(x: Sequence[int]) -> None:
    # Type of x is Sequence[int] here; we don't know the concrete type.
    x = list(x)
    # Type of x is List[int] here.
    x.sort()  # Okay!
```

23.7 Invariance vs covariance

Most mutable generic collections are invariant, and mypy considers all user-defined generic classes invariant by default (see *Variance of generic types* for motivation). This could lead to some unexpected errors when combined with type inference. For example:

```
class A: ...
class B(A): ...

lst = [A(), A()]  # Inferred type is List[A]
new_lst = [B(), B()]  # inferred type is List[B]
lst = new_lst  # mypy will complain about this, because List is invariant
```

Possible strategies in such situations are:

- Use an explicit type annotation:

```
new_lst: List[A] = [B(), B()]
lst = new_lst  # OK
```

- Make a copy of the right hand side:

```
lst = list(new_lst) # Also OK
```

- Use immutable collections as annotations whenever possible:

```
def f_bad(x: List[A]) -> A:
    return x[0]
f_bad(new_lst) # Fails

def f_good(x: Sequence[A]) -> A:
    return x[0]
f_good(new_lst) # OK
```

23.8 Covariant subtyping of mutable protocol members is rejected

Mypy rejects this because this is potentially unsafe. Consider this example:

```
from typing_extensions import Protocol

class P(Protocol):
    x: float

def fun(arg: P) -> None:
    arg.x = 3.14

class C:
    x = 42
c = C()
fun(c)  # This is not safe
c.x << 5  # Since this will fail!
```

To work around this problem consider whether "mutating" is actually part of a protocol. If not, then one can use a `@property` in the protocol definition:

```
from typing_extensions import Protocol

class P(Protocol):
    @property
    def x(self) -> float:
       pass

def fun(arg: P) -> None:
    ...

class C:
    x = 42
fun(C())  # OK
```

23.9 Declaring a supertype as variable type

Sometimes the inferred type is a subtype (subclass) of the desired type. The type inference uses the first assignment to infer the type of a name (assume here that `Shape` is the base class of both `Circle` and `Triangle`):

```
shape = Circle()    # Infer shape to be Circle
...
shape = Triangle()  # Type error: Triangle is not a Circle
```

You can just give an explicit type for the variable in cases such the above example:

```
shape = Circle() # type: Shape   # The variable s can be any Shape,
                                 # not just Circle
...
shape = Triangle()               # OK
```

23.10 Complex type tests

Mypy can usually infer the types correctly when using `isinstance()` type tests, but for other kinds of checks you may need to add an explicit type cast:

```
def f(o: object) -> None:
    if type(o) is int:
        o = cast(int, o)
        g(o + 1)    # This would be an error without the cast
        ...
    else:
        ...
```

Note: Note that the `object` type used in the above example is similar to `Object` in Java: it only supports operations defined for *all* objects, such as equality and `isinstance()`. The type `Any`, in contrast, supports all operations, even if they may fail at runtime. The cast above would have been unnecessary if the type of `o` was `Any`.

Mypy can't infer the type of `o` after the `type()` check because it only knows about `isinstance()` (and the latter is better style anyway). We can write the above code without a cast by using `isinstance()`:

```
def f(o: object) -> None:
    if isinstance(o, int):  # Mypy understands isinstance checks
        g(o + 1)        # Okay; type of o is inferred as int here
        ...
```

Type inference in mypy is designed to work well in common cases, to be predictable and to let the type checker give useful error messages. More powerful type inference strategies often have complex and difficult-to-predict failure modes and could result in very confusing error messages. The tradeoff is that you as a programmer sometimes have to give the type checker a little help.

23.11 Python version and system platform checks

Mypy supports the ability to perform Python version checks and platform checks (e.g. Windows vs Posix), ignoring code paths that won't be run on the targeted Python version or platform. This allows you to more effectively typecheck code that supports multiple versions of Python or multiple operating systems.

More specifically, mypy will understand the use of `sys.version_info` and `sys.platform` checks within `if/elif/else` statements. For example:

```
import sys

# Distinguishing between different versions of Python:
if sys.version_info >= (3, 5):
    # Python 3.5+ specific definitions and imports
elif sys.version_info[0] >= 3:
    # Python 3 specific definitions and imports
else:
    # Python 2 specific definitions and imports

# Distinguishing between different operating systems:
if sys.platform.startswith("linux"):
    # Linux-specific code
elif sys.platform == "darwin":
    # Mac-specific code
elif sys.platform == "win32":
    # Windows-specific code
else:
    # Other systems
```

Note: Mypy currently does not support more complex checks, and does not assign any special meaning when assigning a `sys.version_info` or `sys.platform` check to a variable. This may change in future versions of mypy.

By default, mypy will use your current version of Python and your current operating system as default values for `sys.version_info` and `sys.platform`.

To target a different Python version, use the `--python-version X.Y` flag. For example, to verify your code typechecks if were run using Python 2, pass in `--python-version 2.7` from the command line. Note that you do not need to have Python 2.7 installed to perform this check.

To target a different operating system, use the `--platform PLATFORM` flag. For example, to verify your code typechecks if it were run in Windows, pass in `--platform win32`. See the documentation for sys.platform for examples of valid platform parameters.

23.12 Displaying the type of an expression

You can use `reveal_type(expr)` to ask mypy to display the inferred static type of an expression. This can be useful when you don't quite understand how mypy handles a particular piece of code. Example:

```
reveal_type((1, 'hello'))  # Revealed type is 'Tuple[builtins.int, builtins.str]'
```

You can also use `reveal_locals()` at any line in a file to see the types of all local varaibles at once. Example:

```
a = 1
b = 'one'
reveal_locals()
# Revealed local types are:
# a: builtins.int
# b: builtins.str
```

Note: `reveal_type` and `reveal_locals` are only understood by mypy and don't exist in Python. If you try to run your program, you'll have to remove any `reveal_type` and `reveal_locals` calls before you can run your code. Both are always available and you don't need to import them.

23.13 Import cycles

An import cycle occurs where module A imports module B and module B imports module A (perhaps indirectly, e.g. `A -> B -> C -> A`). Sometimes in order to add type annotations you have to add extra imports to a module and those imports cause cycles that didn't exist before. If those cycles become a problem when running your program, there's a trick: if the import is only needed for type annotations in forward references (string literals) or comments, you can write the imports inside `if TYPE_CHECKING:` so that they are not executed at runtime. Example:

File `foo.py`:

```
from typing import List, TYPE_CHECKING

if TYPE_CHECKING:
    import bar

def listify(arg: 'bar.BarClass') -> 'List[bar.BarClass]':
    return [arg]
```

File `bar.py`:

```
from typing import List
from foo import listify

class BarClass:
    def listifyme(self) -> 'List[BarClass]':
        return listify(self)
```

Note: The `TYPE_CHECKING` constant defined by the `typing` module is `False` at runtime but `True` while type checking.

Python 3.5.1 doesn't have `typing.TYPE_CHECKING`. An alternative is to define a constant named `MYPY` that has the value `False` at runtime. Mypy considers it to be `True` when type checking. Here's the above example modified to use `MYPY`:

```
from typing import List

MYPY = False
if MYPY:
    import bar

def listify(arg: 'bar.BarClass') -> 'List[bar.BarClass]':
    return [arg]
```

23.14 Silencing linters

In some cases, linters will complain about unused imports or code. In these cases, you can silence them with a comment after type comments, or on the same line as the import:

```
# to silence complaints about unused imports
from typing import List  # noqa
a = None  # type: List[int]
```

To silence the linter on the same line as a type comment put the linter comment *after* the type comment:

```
a = some_complex_thing()  # type: ignore  # noqa
```

23.15 Dealing with conflicting names

Suppose you have a class with a method whose name is the same as an imported (or built-in) type, and you want to use the type in another method signature. E.g.:

```
class Message:
    def bytes(self):
        ...
    def register(self, path: bytes):  # error: Invalid type "mod.Message.bytes"
        ...
```

The third line elicits an error because mypy sees the argument type `bytes` as a reference to the method by that name. Other than renaming the method, a work-around is to use an alias:

```
bytes_ = bytes
class Message:
    def bytes(self):
        ...
    def register(self, path: bytes_):
        ...
```

CHAPTER 24

Frequently Asked Questions

24.1 Why have both dynamic and static typing?

Dynamic typing can be flexible, powerful, convenient and easy. But it's not always the best approach; there are good reasons why many developers choose to use statically typed languages.

Here are some potential benefits of mypy-style static typing:

- Static typing can make programs easier to understand and maintain. Type declarations can serve as machine-checked documentation. This is important as code is typically read much more often than modified, and this is especially important for large and complex programs.
- Static typing can help you find bugs earlier and with less testing and debugging. Especially in large and complex projects this can be a major time-saver.
- Static typing can help you find difficult-to-find bugs before your code goes into production. This can improve reliability and reduce the number of security issues.
- Static typing makes it practical to build very useful development tools that can improve programming productivity or software quality, including IDEs with precise and reliable code completion, static analysis tools, etc.
- You can get the benefits of both dynamic and static typing in a single language. Dynamic typing can be perfect for a small project or for writing the UI of your program, for example. As your program grows, you can adapt tricky application logic to static typing to help maintenance.

See also the front page of the mypy web site.

24.2 Would my project benefit from static typing?

For many projects dynamic typing is perfectly fine (we think that Python is a great language). But sometimes your projects demand bigger guns, and that's when mypy may come in handy.

If some of these ring true for your projects, mypy (and static typing) may be useful:

- Your project is large or complex.

- Your codebase must be maintained for a long time.
- Multiple developers are working on the same code.
- Running tests takes a lot of time or work (type checking may help you find errors early in development, reducing the number of testing iterations).
- Some project members (devs or management) don't like dynamic typing, but others prefer dynamic typing and Python syntax. Mypy could be a solution that everybody finds easy to accept.
- You want to future-proof your project even if currently none of the above really apply.

24.3 Can I use mypy to type check my existing Python code?

It depends. Compatibility is pretty good, but some Python features are not yet implemented or fully supported. The ultimate goal is to make using mypy practical for most Python code. Code that uses complex introspection or metaprogramming may be impractical to type check, but it should still be possible to use static typing in other parts of a program.

24.4 Will static typing make my programs run faster?

Mypy only does static type checking and it does not improve performance. It has a minimal performance impact. In the future, there could be other tools that can compile statically typed mypy code to C modules or to efficient JVM bytecode, for example, but this is outside the scope of the mypy project. It may also be possible to modify existing Python VMs to take advantage of static type information, but whether this is feasible is still unknown. This is nontrivial since the runtime types do not necessarily correspond to the static types.

24.5 How do I type check my Python 2 code?

You can use a comment-based function annotation syntax and use the `--py2` command-line option to type check your Python 2 code. You'll also need to install `typing` for Python 2 via `pip install typing`.

24.6 Is mypy free?

Yes. Mypy is free software, and it can also be used for commercial and proprietary projects. Mypy is available under the MIT license.

24.7 Can I use structural subtyping?

Mypy provides support for both nominal subtyping and structural subtyping. Some argue that structural subtyping is better suited for languages with duck typing such as Python. Mypy however primarily uses nominal subtyping, leaving structural subtyping mostly opt-in (except for built-in protocols such as `Iterable` that always support structural subtyping). Here are some reasons why:

1. It is easy to generate short and informative error messages when using a nominal type system. This is especially important when using type inference.
2. Python provides built-in support for nominal `isinstance()` tests and they are widely used in programs. Only limited support for structural `isinstance()` is available, and it's less type safe than nominal type tests.

3. Many programmers are already familiar with static, nominal subtyping and it has been successfully used in languages such as Java, C++ and C#. Fewer languages use structural subtyping.

However, structural subtyping can also be useful. For example, a "public API" may be more flexible if it is typed with protocols. Also, using protocol types removes the necessity to explicitly declare implementations of ABCs. As a rule of thumb, we recommend using nominal classes where possible, and protocols where necessary. For more details about protocol types and structural subtyping see *Protocols and structural subtyping* and PEP 544.

24.8 I like Python and I have no need for static typing

That wasn't really a question, was it? Mypy is not aimed at replacing Python. The goal is to give more options for Python programmers, to make Python a more competitive alternative to other statically typed languages in large projects, to improve programmer productivity and to improve software quality.

24.9 How are mypy programs different from normal Python?

Since you use a vanilla Python implementation to run mypy programs, mypy programs are also Python programs. The type checker may give warnings for some valid Python code, but the code is still always runnable. Also, some Python features and syntax are still not supported by mypy, but this is gradually improving.

The obvious difference is the availability of static type checking. The section *Common issues* mentions some modifications to Python code that may be required to make code type check without errors. Also, your code must make attributes explicit and use a explicit protocol representation. For example, you may want to subclass an Abstract Base Class such as `typing.Iterable`.

Mypy will support modular, efficient type checking, and this seems to rule out type checking some language features, such as arbitrary runtime addition of methods. However, it is likely that many of these features will be supported in a restricted form (for example, runtime modification is only supported for classes or methods registered as dynamic or 'patchable').

24.10 How is mypy different from Cython?

Cython is a variant of Python that supports compilation to CPython C modules. It can give major speedups to certain classes of programs compared to CPython, and it provides static typing (though this is different from mypy). Mypy differs in the following aspects, among others:

- Cython is much more focused on performance than mypy. Mypy is only about static type checking, and increasing performance is not a direct goal.
- The mypy syntax is arguably simpler and more "Pythonic" (no cdef/cpdef, etc.) for statically typed code.
- The mypy syntax is compatible with Python. Mypy programs are normal Python programs that can be run using any Python implementation. Cython has many incompatible extensions to Python syntax, and Cython programs generally cannot be run without first compiling them to CPython extension modules via C. Cython also has a pure Python mode, but it seems to support only a subset of Cython functionality, and the syntax is quite verbose.
- Mypy has a different set of type system features. For example, mypy has genericity (parametric polymorphism), function types and bidirectional type inference, which are not supported by Cython. (Cython has fused types that are different but related to mypy generics. Mypy also has a similar feature as an extension of generics.)
- The mypy type checker knows about the static types of many Python stdlib modules and can effectively type check code that uses them.

- Cython supports accessing C functions directly and many features are defined in terms of translating them to C or C++. Mypy just uses Python semantics, and mypy does not deal with accessing C library functionality.

24.11 How is mypy different from Nuitka?

Nuitka is a static compiler that can translate Python programs to C++. Nuitka integrates with the CPython runtime. Nuitka has additional future goals, such as using type inference and whole-program analysis to further speed up code. Here are some differences:

- Nuitka is primarily focused on speeding up Python code. Mypy focuses on static type checking and facilitating better tools.
- Whole-program analysis tends to be slow and scale poorly to large or complex programs. It is still unclear if Nuitka can solve these issues. Mypy does not use whole-program analysis and will support modular type checking (though this has not been implemented yet).

24.12 How is mypy different from RPython or Shed Skin?

RPython and Shed Skin are basically statically typed subsets of Python. Mypy does the following important things differently:

- RPython is primarily designed for implementing virtual machines; mypy is a general-purpose tool.
- Mypy supports both static and dynamic typing. Dynamically typed and statically typed code can be freely mixed and can interact seamlessly.
- Mypy aims to support (in the future) fast and modular type checking. Both RPython and Shed Skin use whole-program type inference which is very slow, does not scale well to large programs and often produces confusing error messages. Mypy can support modularity since it only uses local type inference; static type checking depends on having type annotations for functions signatures.
- Mypy will support introspection, dynamic loading of code and many other dynamic language features (though using these may make static typing less effective). RPython and Shed Skin only support a restricted Python subset without several of these features.
- Mypy supports user-defined generic types.

24.13 Mypy is a cool project. Can I help?

Any help is much appreciated! Contact the developers if you would like to contribute. Any help related to development, design, publicity, documentation, testing, web site maintenance, financing, etc. can be helpful. You can learn a lot by contributing, and anybody can help, even beginners! However, some knowledge of compilers and/or type systems is essential if you want to work on mypy internals.

CHAPTER 25

Revision history

List of major changes (the Mypy Blog contains more detailed release notes):

- **May 2018**
  - Publish `mypy` version 0.600 on PyPI.
    - Enable *strict optional checking* by default.
    - Document *disabling strict optional checking*.
    - Add *Mypy daemon (mypy server)*.
    - Add *Using a remote cache to speed up mypy runs*.
    - Support user-specific configuration file (*docs*).
    - Changes to section pattern semantics in configuration files (*docs* and *more docs*).
- **April 2018**
  - Publish `mypy` version 0.590 on PyPI.
    - Document *PEP 561 support*.
    - Made *incremental mode* the default.
    - Document `--always-true` and `--always-false` (*docs*).
    - Document `follow_imports_for_stubs` (*docs*).
    - Add coroutines to *Python 3 cheat sheet*.
    - Add `None` return/strict-optional to *common issues*.
    - Clarify that `SupportsInt` etc. don't support arithmetic operations (see *docs*).
- **March 2018**
  - Publish `mypy` version 0.580 on PyPI.
    - Allow specifying multiple packages on the command line with `-p` and `-m` flags.
  - Publish `mypy` version 0.570 on PyPI.

    - Add support for *The attrs package*.

- **December 2017**
  - Publish `mypy` version 0.560 on PyPI.
    - Various types in `typing` that used to be ABCs *are now protocols* and support *structural subtyping*.
    - Explain how to *silence invalid complaints* by linters about unused imports due to type comments.
- **November 2017**
  - Publish `mypy` version 0.550 on PyPI.
    - Running mypy now requires Python 3.4 or higher. However Python 3.3 is still valid for the target of the analysis (i.e. the `--python-version` flag).
    - Split `--disallow-any` flag into *separate boolean flags*.
    - The `--old-html-report` flag was removed.
- **October 2017**
  - Publish `mypy` version 0.540 on PyPI.
  - Publish `mypy` version 0.530 on PyPI.
- **August-September 2017**
  - Add *Protocols and structural subtyping*.
  - Other updates to *The mypy command line*:
    - Add `--warn-unused-configs`.
    - Add `--disallow-untyped-decorators`.
    - Add `--disallow-incomplete-defs`.
- **July 2017**
  - Publish `mypy` version 0.521 on PyPI.
  - Publish `mypy` version 0.520 on PyPI.
  - Add *fine-grained control of Any types*.
  - Add *TypedDict*.
  - Other updates to *The mypy command line*:
    - Add `--no-implicit-optional`.
    - Add `--shadow-file`.
    - Add `--no-incremental`.
- **May 2017**
  - Publish `mypy` version 0.510 on PyPI.
  - Remove option `--no-fast-parser`.
  - Deprecate option `--strict-boolean`.
  - Drop support for Python 3.2 as type checking target.
  - Add support for *overloaded functions with implementations*.
  - Add *Extended Callable types*.

  - Add *Asynchronous generators (PEP 525) and comprehensions (PEP 530)*.
  - Add *ClassVar*.
  - Add *quick mode*.
- **March 2017**
  - Publish `mypy` version 0.500 on PyPI.
  - Add *The NoReturn type*.
  - Add *Defining sub-classes of generic classes*.
  - Add *Variance of generic types*.
  - Add *Invariance vs covariance*.
  - Updates to *New features in Python 3.6*.
  - Updates to *Integrating mypy into another Python application*.
  - Updates to *The mypy command line*:
    - Add option `--warn-return-any`.
    - Add option `--strict-boolean`.
    - Add option `--strict`.
  - Updates to *The mypy configuration file*:
    - `warn_no_return` is on by default.
    - Read settings from `setup.cfg` if `mypy.ini` does not exist.
    - Add option `warn_return_any`.
    - Add option `strict_boolean`.
- **January 2017**
  - Publish `mypy` version 0.470 on PyPI.
  - Change package name from `mypy-lang` to `mypy`.
  - Add *Integrating mypy into another Python application*.
  - Add *Type hints cheat sheet (Python 3)*.
  - Major update to *How imports are found*.
  - Add *–ignore-missing-imports*.
  - Updates to *The mypy configuration file*.
  - Document underscore support in numeric literals.
  - Document that arguments prefixed with ___ are positional-only.
  - Document that `--hide-error-context` is now on by default, and there is a new flag `--show-error-context`.
  - Add `ignore_errors` to *Per-module flags*.
- **November 2016**
  - Publish `mypy-lang` version 0.4.6 on PyPI.
  - Add *Getting started*.

  - Add *Generic methods and generic self* (experimental).
  - Add *Declaring decorators*.
  - Discuss generic type aliases in *Type aliases*.
  - Discuss Python 3.6 named tuple syntax in *Named tuples*.
  - Updates to *Common issues*.
  - Updates to *New features in Python 3.6*.
  - Updates to *The mypy command line*:
    * `--custom-typeshed-dir`
    * `--junit-xml`
    * `--find-occurrences`
    * `--cobertura-xml-report`
    * `--warn-no-return`
  - Updates to *The mypy configuration file*:
    * Sections with fnmatch patterns now use module name patterns (previously they were path patterns).
    * Added `custom_typeshed_dir`, `mypy_path` and `show_column_numbers`.
  - Mention the magic `MYPY` constant in *Import cycles*.

- **October 2016**
  - Publish `mypy-lang` version 0.4.5 on PyPI.
  - Add *New features in Python 3.6*.
  - Add *The mypy configuration file*.
  - Updates to *The mypy command line*: `--strict-optional-white-list`, `--disallow-subclassing-any`, `--config-file`, `@flagfile`, `--hide-error-context` (replaces `--suppress-error-context`), `--show-column-numbers` and `--scripts-are-modules`.
  - Mention `typing.TYPE_CHECKING` in *Import cycles*.
- **August 2016**
  - Publish `mypy-lang` version 0.4.4 on PyPI.
  - Add *NewTypes*.
  - Add *Typing async/await*.
  - Add *Text and AnyStr*.
  - Add *Python version and system platform checks*.
- **July 2016**
  - Publish `mypy-lang` version 0.4.3 on PyPI.
  - Add *strict optional checking*.
  - Add *Multi-line Python 2 function annotations*.
- **June 2016**

  - Publish `mypy-lang` version 0.4.2 on PyPI.
  - Add *The type of class objects*.
  - Add *Type hints cheat sheet (Python 2)*.
  - Add *Displaying the type of an expression*.
- **May 2016**
  - Publish `mypy-lang` version 0.4 on PyPI.
  - Add *Type variables with upper bounds*.
  - Document *The mypy command line*.
- **Feb 2016**
  - Publish `mypy-lang` version 0.3.1 on PyPI.
  - Document Python 2 support.
- **Nov 2015** Add *Library stubs and the Typeshed repo*.
- **Jun 2015** Remove `Undefined` and `Dynamic`, as they are not in PEP 484.
- **Apr 2015** Publish `mypy-lang` version 0.2.0 on PyPI.
- **Mar 2015** Update documentation to reflect PEP 484:
  - Add *Named tuples* and *Optional types*.
  - Do not mention type application syntax (for example, `List[int]()`), as it's no longer supported, due to PEP 484 compatibility.
  - Rename `typevar` to `TypeVar`.
  - Document `# type: ignore` which allows locally ignoring spurious errors (*Spurious errors and locally silencing the checker*).
  - No longer mention `Any(x)` as a valid cast, as it will be phased out soon.
  - Mention the new `.pyi` stub file extension. Stubs can live in the same directory as the rest of the program.
- **Jan 2015** Mypy moves closer to PEP 484:
  - Add *Type aliases*.
  - Update discussion of overloading – it's now only supported in stubs.
  - Rename `Function[...]` to `Callable[...]`.
- **Dec 2014** Publish mypy version 0.1.0 on PyPI.
- **Oct 2014** Major restructuring. Split the HTML documentation into multiple pages.
- **Sep 2014** Migrated docs to Sphinx.
- **Aug 2014** Don't discuss native semantics. There is only Python semantics.
- **Jul 2013** Rewrite to use new syntax. Shift focus to discussing Python semantics. Add more content, including short discussions of *Generic functions* and *Union types*.

CHAPTER 26

Indices and tables

- genindex
- search