Mypy Documentation

Release 0.680+dev.2814f360af80a6ea3b5aac560e4d0dab1ccd5e35.dirty

Jukka

Mar 20, 2019

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, callable types, tuple types, union types, and structural subtyping.

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

Getting started

This chapter introduces some core concepts of mypy, including function annotations, the `typing` module, library stubs, and more.

Be sure to read this chapter carefully, as the rest of the documentation may not make much sense otherwise.

2.1 Installing and running mypy

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

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

Once mypy is installed, run it by using the `mypy` tool:

```
$ mypy program.py
```

This command makes mypy *type check* your `program.py` file and print out any errors it finds. Mypy will type check your code *statically*: this means that it will check for errors without ever running your code, just like a linter.

This means that you are always free to ignore the errors mypy reports and treat them as just warnings, if you so wish: mypy runs independently from Python itself.

However, if you try directly running mypy on your existing Python code, it will most likely report little to no errors: you must add *type annotations* to your code to take full advantage of mypy. See the section below for details.

Note: Although you must install Python 3 to run mypy, mypy is fully capable of type checking Python 2 code as well: just pass in the `--py2` flag. See *Type checking Python 2 code* for more details.

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

2.2 Function signatures and dynamic vs static typing

A function without type annotations is considered to be *dynamically typed* by mypy:

```
def greeting(name):
    return 'Hello ' + name
```

By default, mypy will **not** type check dynamically typed functions. This means that with a few exceptions, mypy will not report any errors with regular unannotated Python.

This is the case even if you misuse the function: for example, mypy would currently not report any errors if you tried running `greeting(3)` or `greeting(b"Alice")` even though those function calls would result in errors at runtime.

You can teach mypy to detect these kinds of bugs by adding *type annotations* (also known as *type hints*). For example, you can teach mypy that `greeting` both accepts and returns a string like so:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

This function is now *statically typed*: mypy can use the provided type hints to detect incorrect usages of the `greeting` function. For example, it will reject the following calls since the arguments have invalid types:

```
def greeting(name: str) -> str:
    return 'Hello ' + name

greeting(3)          # Argument 1 to "greeting" has incompatible type "int"; expected
↪"str"
greeting(b'Alice')   # Argument 1 to "greeting" has incompatible type "bytes";␣
↪expected "str"
```

Note that this is all still valid Python 3 code! The function annotation syntax shown above was added to Python as a part of Python 3.0.

If you are trying to type check Python 2 code, you can add type hints using a comment-based syntax instead of the Python 3 annotation syntax. See our section on *typing Python 2 code* for more details.

Being able to pick whether you want a function to be dynamically or statically typed can be very helpful. For example, if you are migrating an existing Python codebase to use static types, it's usually easier to migrate by incrementally adding type hints to your code rather than adding them all at once. Similarly, when you are prototyping a new feature, it may be convenient to initially implement the code using dynamic typing and only add type hints later once the code is more stable.

Once you are finished migrating or prototyping your code, you can make mypy warn you if you add a dynamic function by mistake by using the `--disallow-untyped-defs` flag. See *The mypy command line* for more information on configuring mypy.

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

2.3 More function signatures

Here are a few more examples of adding type hints to function signatures.

If a function does not explicitly return a value, give it a return type of `None`. Using a `None` result in a statically typed context results in a type check error:

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

a = p()  # Error: "p" does not return a value
```

Make sure to remember to include `None`: if you don't, the function will be dynamically typed. For example:

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

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

Arguments with default values can be annotated like so:

```
def greeting(name: str, excited: bool = False) -> str:
    message = 'Hello, {}'.format(name)
    if excited:
        message += '!!!'
    return message
```

`*args` and `**kwargs` arguments can be annotated like so:

```
def stars(*args: int, **kwargs: float) -> None:
    # 'args' has type 'Tuple[int, ...]' (a tuple of ints)
    # 'kwargs' has type 'Dict[str, float]' (a dict of strs to floats)
    for arg in args:
        print(name)
    for key, value in kwargs:
        print(key, value)
```

2.4 The typing module

So far, we've added type hints that use only basic concrete types like `str` and `float`. What if we want to express more complex types, such as "a list of strings" or "an iterable of ints"?

You can find many of these more complex static types inside of the `typing` module. For example, to indicate that some function can accept a list of strings, use the `List` type from the `typing` module:

```
from typing import List

def greet_all(names: List[str]) -> None:
    for name in names:
        print('Hello ' + name)

names = ["Alice", "Bob", "Charlie"]
ages = [10, 20, 30]

greet_all(names)    # Ok!
greet_all(ages)     # Error due to incompatible types
```

The `List` type is an example of something called a *generic type*: it can accept one or more *type parameters*. In this case, we *parameterized* `List` by writing `List[str]`. This lets mypy know that `greet_all` accepts specifically

lists containing strings, and not lists containing ints or any other type.

In this particular case, the type signature is perhaps a little too rigid. After all, there's no reason why this function must accept *specifically* a list – it would run just fine if you were to pass in a tuple, a set, or any other custom iterable.

You can express this idea using the `Iterable` type instead of `List`:

```
from typing import Iterable

def greet_all(names: Iterable[str]) -> None:
    for name in names:
        print('Hello ' + name)
```

As another example, suppose you want to write a function that can accept *either* ints or strings, but no other types. You can express this using the `Union` type:

```
from typing import Union

def normalize_id(user_id: Union[int, str]) -> str:
    if isinstance(user_id, int):
        return 'user-{}'.format(100000 + user_id)
    else:
        return user_id
```

Similarly, suppose that you want the function to accept only strings or None. You can again use `Union` and use `Union[str, None]` – or alternatively, use the type `Optional[str]`. These two types are identical and interchangeable: `Optional[str]` is just a shorthand or *alias* for `Union[str, None]`. It exists mostly as a convenience to help function signatures look a little cleaner:

```
from typing import Optional

def greeting(name: Optional[str] = None) -> str:
    # Optional[str] means the same thing as Union[str, None]
    if name is None:
        name = 'stranger'
    return 'Hello, ' + name
```

The `typing` module contains many other useful types. You can find a quick overview by looking through the *mypy cheatsheets* and a more detailed overview (including information on how to make your own generic types or your own type aliases) by looking through the *type system reference*.

One final note: when adding types, the convention is to import types using the form `from typing import Iterable` (as opposed to doing just `import typing` or `import typing as t` or `from typing import *`).

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

2.5 Local type inference

Once you have added type hints to a function (i.e. made it statically typed), mypy will automatically type check that function's body. While doing so, mypy will try and *infer* as many details as possible.

We saw an example of this in the `normalize_id` function above – mypy understands basic `isinstance` checks and so can infer that the `user_id` variable was of type `int` in the if-branch and of type `str` in the else-branch. Similarly, mypy was able to understand that `name` could not possibly be `None` in the `greeting` function above, based both on the `name is None` check and the variable assignment in that if statement.

As another example, consider the following function. Mypy can type check this function without a problem: it will use the available context and deduce that `output` must be of type `List[float]` and that `num` must be of type `float`:

```
def nums_below(numbers: Iterable[float], limit: float) -> List[float]:
    output = []
    for num in numbers:
        if num < limit:
            output.append(num)
    return output
```

Mypy will warn you if it is unable to determine the type of some variable – for example, when assigning an empty dictionary to some global value:

```
my_global_dict = {}  # Error: Need type annotation for 'my_global_dict'
```

You can teach mypy what type `my_global_dict` is meant to have by giving it a type hint. For example, if you knew this variable is supposed to be a dict of ints to floats, you could annotate it using either variable annotations (introduced in Python 3.6 by **:ref:'PEP 526 <pep526_>'_**) or using a comment-based syntax like so:

```
# If you're using Python 3.6+
my_global_dict: Dict[int, float] = {}

# If you want compatibility with older versions of Python
my_global_dict = {}  # type: Dict[int, float]
```

2.6 Library stubs and typeshed

Mypy uses library *stubs* to type check code interacting with library modules, including the Python standard library. A library stub defines a skeleton of the public interface of the library, including classes, variables and functions, and their types. Mypy ships with stubs from the typeshed project, which contains library stubs for the Python builtins, the standard library, and selected third-party packages.

For example, consider this code:

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

Without a library stub, mypy would have no way of inferring the type of `x` and checking that the argument to `chr` has a valid type.

Mypy complains if it can't find a stub (or a real module) for a library module that you import. Some modules ship with stubs that mypy can automatically find, or you can install a 3rd party module with additional stubs (see *Using installed packages* for details). You can also *create stubs* easily. We discuss ways of silencing complaints about missing stubs in *Missing imports*.

2.7 Configuring mypy

Mypy supports many command line options that you can use to tweak how mypy behaves: see *The mypy command line* for more details.

For example, suppose you want to make sure *all* functions within your codebase are using static typing and make mypy report an error if you add a dynamically-typed function by mistake. You can make mypy do this by running mypy with the `--disallow-untyped-defs` flag.

Another potentially useful flag is `--strict`, which enables many (thought not all) of the available strictness options – including `--disallow-untyped-defs`.

This flag is mostly useful if you're starting a new project from scratch and want to maintain a high degree of type safety from day one. However, this flag will probably be too aggressive if you either plan on using many untyped third party libraries or are trying to add static types to a large, existing codebase. See *Using mypy with an existing codebase* for more suggestions on how to handle the latter case.

2.8 Next steps

If you are in a hurry and don't want to read lots of documentation before getting started, here are some pointers to quick learning resources:

- Read the *mypy cheatsheet* (also for *Python 2*).
- Read *Using mypy with an existing codebase* if you have a significant existing codebase without many type annotations.
- Read the blog post about the Zulip project's experiences with adopting mypy.
- If you prefer watching talks instead of reading, here are some ideas:
  - Carl Meyer: Type Checked Python in the Real World (PyCon 2018)
  - Greg Price: Clearer Code at Scale: Static Types at Zulip and Dropbox (PyCon 2018)
- Look at *solutions to common issues* with mypy if you encounter problems.
- You can ask questions about mypy in the mypy issue tracker and typing Gitter chat.

You can also continue reading this document and skip sections that aren't relevant for you. You don't need to read sections in order.

CHAPTER 3

Using mypy with an existing codebase

This section explains how to get started using mypy with an existing, significant codebase that has little or no type annotations. If you are a beginner, you can skip this section.

These steps will get you started with mypy on an existing codebase:

1. Start small – get a clean mypy build for some files, with few annotations
2. Write a mypy runner script to ensure consistent results
3. Run mypy in Continuous Integration to prevent type errors
4. Gradually annotate commonly imported modules
5. Write annotations as you modify existing code and write new code
6. Use MonkeyType or PyAnnotate to automatically annotate legacy code

We discuss all of these points in some detail below, and a few optional follow-up steps.

3.1 Start small

If your codebase is large, pick a subset of your codebase (say, 5,000 to 50,000 lines) and run mypy only on this subset at first, *without any annotations*. This shouldn't take more than a day or two to implement, so you start enjoying benefits soon.

You'll likely need to fix some mypy errors, either by inserting annotations requested by mypy or by adding `# type: ignore` comments to silence errors you don't want to fix now.

In particular, mypy often generates errors about modules that it can't find or that don't have stub files:

```
core/config.py:7: error: Cannot find module named 'frobnicate'
core/model.py:9: error: Cannot find module named 'acme'
...
```

This is normal, and you can easily ignore these errors. For example, here we ignore an error about a third-party module `frobnicate` that doesn't have stubs using `# type: ignore`:

```
import frobnicate  # type: ignore
...
frobnicate.initialize()  # OK (but not checked)
```

You can also use a mypy configuration file, which is convenient if there are a large number of errors to ignore. For example, to disable errors about importing `frobnicate` and `acme` everywhere in your codebase, use a config like this:

```
[mypy-frobnicate.*]
ignore_missing_imports = True

[mypy-acme.*]
ignore_missing_imports = True
```

You can add multiple sections for different modules that should be ignored.

If your config file is named `mypy.ini`, this is how you run mypy:

```
mypy --config-file mypy.ini mycode/
```

If you get a large number of errors, you may want to ignore all errors about missing imports. This can easily cause problems later on and hide real errors, and it's only recommended as a last resort. For more details, look *here*.

Mypy follows imports by default. This can result in a few files passed on the command line causing mypy to process a large number of imported files, resulting in lots of errors you don't want to deal with at the moment. There is a config file option to disable this behavior, but since this can hide errors, it's not recommended for most users.

3.2 Mypy runner script

Introduce a mypy runner script that runs mypy, so that every developer will use mypy consistently. Here are some things you may want to do in the script:

- Ensure that the correct version of mypy is installed.
- Specify mypy config file or command-line options.
- Provide set of files to type check. You may want to implement inclusion and exclusion filters for full control of the file list.

3.3 Continuous Integration

Once you have a clean mypy run and a runner script for a part of your codebase, set up your Continuous Integration (CI) system to run mypy to ensure that developers won't introduce bad annotations. A simple CI script could look something like this:

```
python3 -m pip install mypy==0.600  # Pinned version avoids surprises
scripts/mypy  # Runs with the correct options
```

3.4 Annotate widely imported modules

Most projects have some widely imported modules, such as utilities or model classes. It's a good idea to annotate these pretty early on, since this allows code using these modules to be type checked more effectively. Since mypy supports

gradual typing, it's okay to leave some of these modules unannotated. The more you annotate, the more useful mypy will be, but even a little annotation coverage is useful.

3.5 Write annotations as you go

Now you are ready to include type annotations in your development workflows. Consider adding something like these in your code style conventions:

1. Developers should add annotations for any new code.
2. It's also encouraged to write annotations when you modify existing code.

This way you'll gradually increase annotation coverage in your codebase without much effort.

3.6 Automate annotation of legacy code

There are tools for automatically adding draft annotations based on type profiles collected at runtime. Tools include MonkeyType (Python 3) and PyAnnotate (type comments only).

A simple approach is to collect types from test runs. This may work well if your test coverage is good (and if your tests aren't very slow).

Another approach is to enable type collection for a small, random fraction of production network requests. This clearly requires more care, as type collection could impact the reliability or the performance of your service.

3.7 Speed up mypy runs

You can use *mypy daemon* to get much faster incremental mypy runs. The larger your project is, the more useful this will be. If your project has at least 100,000 lines of code or so, you may also want to set up *remote caching* for further speedups.

3.8 Introduce stricter options

Mypy is very configurable. Once you get started with static typing, you may want to explore the various strictness options mypy provides to catch more bugs. For example, you can ask mypy to require annotations for all functions in certain modules to avoid accidentally introducing code that won't be type checked. Refer to *The mypy command line* for the details.

CHAPTER 4

Type hints cheat sheet (Python 3)

This document is a quick cheat sheet showing how the PEP 484 type annotation notation represents various common types in 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 Variables

Python 3.6 introduced a syntax for annotating variables in PEP 526 and we use it in most examples.

```
# This is how you declare the type of a variable type in Python 3.6
age: int = 1

# In Python 3.5 and earlier you can use a type comment instead
# (equivalent to the previous definition)
age = 1  # type: int

# You don't need to initialize a variable to annotate it
a: int  # Ok (no value at runtime until assigned)

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

4.2 Built-in types

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

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

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

# Same as above, but with type comment syntax
x = [1]  # type: List[int]

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

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

# Use Optional[] for values that could be None
x: Optional[str] = some_function()
# Mypy understands a value can't be None in an if-statement
if x is not None:
    print(x.upper())
# If a value can never be None due to some invariants, use an assert
assert x is not None
print(x.upper())
```

4.3 Functions

Python 3 supports an annotation syntax for function declarations.

```
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 default value for an argument after the type annotation
def f(num1: int, my_float: float = 3.5) -> float:
    return num1 + my_float

# This is how you annotate a callable (function) value
x: Callable[[int, float], float] = f
```

(continues on next page)

(continued from previous page)

```
# 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

# You can of course split a function annotation over multiple lines
def send_email(address: Union[str, List[str]],
               sender: str,
               cc: Optional[List[str]],
               bcc: Optional[List[str]],
               subject='',
               body: Optional[List[str]] = None
               ) -> bool:
    ...

# 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
```

4.4 When you're puzzled or when things are complicated

```
from typing import Union, Any, List, Optional, 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)  # -> Revealed type is 'builtins.int'

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

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

# If you initialize a variable with an empty container or "None"
# you may have to help mypy a bit by providing a type annotation
x: List[str] = []
x: Optional[str] = None

# 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 a "type: ignore" comment to suppress errors on a given line,
```

(continues on next page)

(continued from previous page)

```
# 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 that lets you override the inferred
# type of an expression. It's only for mypy -- there's no runtime check.
a = [4]
b = cast(List[int], a)  # Passes fine
c = cast(List[str], a)  # Passes fine (no runtime check)
reveal_type(c)  # -> 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"
    # (use "value: Any" to allow arbitrary types)
    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
```

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

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

(continues on next page)

(continued from previous page)

```
    return set(my_mapping.values())

f({3: 'yes', 4: 'no'})
```

4.6 Classes

```
class MyClass:
    # You can optionally declare instance variables in the class body
    attr: int
    # This is an instance variable with a default value
    charge_percent: int = 100

    # 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 type for "self"
    def my_method(self, num: int, str1: str) -> str:
        return num * str1

# User-defined classes are valid as types in annotations
x: MyClass = MyClass()

# You can use the ClassVar annotation to declare a class 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] = []
```

4.7 Coroutines and asyncio

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

```
import asyncio

# A 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!"
```

4.8 Miscellaneous

```
import sys
import re
from typing import Match, AnyStr, IO

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

# Use IO[] for functions that should accept or return any
# object that comes from an open() call (IO[] does not
# distinguish between reading, writing or other modes)
def get_sys_IO(mode: str = '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 defined
def f(foo: A) -> int:  # This will fail
    ...

class A:
    ...

# If you use the string literal 'A', it will pass as long as there is a
# class of that name later on in the file
def f(foo: 'A') -> int:  # Ok
    ...
```

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 = {6, 7}  # type: Set[int]

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

# 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
```

(continues on next page)

(continued from previous page)

```
# ("Text" means  "unicode" in Python 2 and "str" in Python 3)
x = [u"one", u"two"]  # type: List[Text]

# Use Optional[] for values that could be None
x = some_function()  # type: Optional[str]
# Mypy understands a value can't be None in an if-statement
if x is not None:
    print x.upper()
# If a value can never be None due to some invariants, use an assert
assert x is not None
print x.upper()
```

5.2 Functions

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

# 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 arguments with default values as though they
# had no defaults
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 callable (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]
```

(continues on next page)

(continued from previous page)

```
    i = 0
    while i < n:
        yield i
        i += 1

# There's an 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, List, Optional, 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) # -> 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

# If you initialize a variable with an empty container or "None"
# you may have to help mypy a bit by providing a type annotation
x = []  # type: List[str]
x = None  # type: Optional[str]

# 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 a "type: ignore" comment to suppress errors 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 that lets you override the inferred
# type of an expression. It's only for mypy -- there's no runtime check.
a = [4]
b = cast(List[int], a)  # Passes fine
c = cast(List[str], a)  # Passes fine (no runtime check)
```

(continues on next page)

(continued from previous page)

```
reveal_type(c)  # -> 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"
    # (use "value: Any" to allow arbitrary types)
    def __setattr__(self, name, value):
        # type: (str, int) -> None
        ...

a.foo = 42  # Works
a.bar = 'Ex-parrot'  # Fails type checking
```

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_mapping[5] = 'maybe'
    return set(my_mapping.values())

f({3: 'yes', 4: 'no'})
```

5.5 Classes

```
class MyClass(object):
    # For instance methods, omit type for "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):
        # type: () -> None
        pass

# User-defined classes are valid as types in annotations
x = MyClass()  # type: MyClass
```

5.6 Miscellaneous

```
import sys
import re
from typing import Match, AnyStr, IO

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

# Use IO[] for functions that should accept or return any
# object that comes from an open() call (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
```

CHAPTER 6

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 `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 `collections.abc` (formerly `collections`), they are not identical, since the built-in collection type objects do not support indexing.

CHAPTER 7

Type inference and type annotations

7.1 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.

7.2 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>)`.

7.3 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()
```

7.4 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.

7.5 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)
```

7.6 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
```

7.7 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]`.

CHAPTER 8

Kinds of types

We've mostly restricted ourselves to built-in types until now. This section introduces several additional kinds of types. You are likely to need at least some of them to type check any non-trivial programs.

8.1 Class types

Every class is also a valid type. Any instance of a subclass is also compatible with all superclasses – it follows that every value is compatible with the `object` type (and incidentally also the `Any` type, discussed below). Mypy analyzes the bodies of classes to determine which methods and attributes are available in instances. This example uses subclassing:

```
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

def foo(a: A) -> None:
    print(a.f())  # 3
    a.g()         # Error: "A" has no attribute "g"

foo(B())  # OK (B is a subclass of A)
```

8.2 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 only checked at runtime. You can use

`Any` as an "escape hatch" when you can't use a more precise type for some reason.

`Any` is compatible with every other type, and vice versa. You can freely assign a value of type `Any` to a variable with a more precise type:

```
a: Any = None
s: str = ''
a = 2      # OK (assign "int" to "Any")
s = a      # OK (assign "Any" to "str")
```

Declared (and inferred) types are ignored (or *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, while the declared type of `s` is actually `str`! You need to be careful with `Any` types, since they let you lie to mypy, and this could easily hide bugs.

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.

8.3 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 valid as a base class in Python 3.6 and later, and always in stub files. In earlier Python versions you can sometimes work around this limitation by using a named tuple as a base class (see section *Named tuples*).

8.4 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
```

In situations where more precise or complex types of callbacks are necessary one can use flexible *callback protocols*.

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.

8.5 Union types

Python functions often accept values of two or more different types. You can use *overloading* to represent this, but union types are often more convenient.

Use the `Union[T1, ..., Tn]` type constructor to construct a union type. For example, if an argument has type `Union[int, str]`, both integers and strings are valid argument values.

You can use an `isinstance()` check to narrow down a union type to a more 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:
        # Here type of x is str.
        x + 'a'    # OK

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

Note: Operations are valid for union types only if they are valid for *every* union item. This is why it's often necessary to use an `isinstance()` check to first narrow down a union type to a non-union type. This also means that it's recommended to avoid union types as function return types, since the caller may have to use `isinstance()` before doing anything interesting with the value.

8.6 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:
```

(continues on next page)

(continued from previous page)

```
        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 syntactically 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.

8.7 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.

8.8 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*.)

8.9 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:
    ...
```

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 except for *generic aliases*.

8.10 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 introduced an alternative, class-based syntax for named tuples with types:

```
from typing import NamedTuple

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

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

8.11 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.

8.12 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.

8.13 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`.

CHAPTER 9

Class basics

This section will help get you started annotating your classes. Built-in classes such as `int` also follow these same rules.

9.1 Instance and class attributes

The 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  # Aha, 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 annotation:

```
class A:
    x: List[int]  # Declare attribute 'x' of type List[int]

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

As in Python generally, a variable defined in the class body can be used as a class or an instance variable. (As discussed in the next section, you can override this with a `ClassVar` annotation.)

Type comments work as well, if you need to support Python versions earlier than 3.6:

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

Note that attribute definitions in the class body that use a type comment are special: a `None` value is valid as the initializer, even though the declared type is not optional. This should be used sparingly, as this can result in `None`-related runtime errors that mypy can't detect.

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

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

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

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
```

9.2 Annotating *__init__* methods

The `__init__` method is somewhat special – it doesn't return a value. This is best expressed as `-> None`. However, since many feel this is redundant, it is allowed to omit the return type declaration on `__init__` methods **if at least one argument is annotated**. For example, in the following classes `__init__` is considered fully annotated:

```
class C1:
    def __init__(self) -> None:
        self.var = 42

class C2:
    def __init__(self, arg: int):
        self.var = arg
```

However, if `__init__` has no annotated arguments and no return type annotation, it is considered an untyped method:

```
class C3:
    def __init__(self):
        # This body is not type checked
        self.var = 42 + 'abc'
```

9.3 Class attribute annotations

You can use a `ClassVar[t]` annotation to explicitly declare that a particular attribute should not be set on instances:

```
from typing import ClassVar

class A:
```

(continues on next page)

(continued from previous page)

```
    x: ClassVar[int] = 0  # Class variable only

A.x += 1  # OK

a = A()
a.x = 1  # Error: Cannot assign to class variable "x" via instance
print(a.x)  # OK -- can be read through an instance
```

Note: If you need to support Python 3 versions 3.5.2 or earlier, you have to import `ClassVar` from `typing_extensions` instead (available on PyPI). If you use Python 2.7, you can import it from `typing`.

It's not necessary to annotate all class variables using `ClassVar`. An attribute without the `ClassVar` annotation can still be used as a class variable. However, mypy won't prevent it from being used as an instance variable, as discussed previously:

```
class A:
    x = 0  # Can be used as a class or instance variable

A.x += 1  # OK

a = A()
a.x = 1  # Also OK
```

Note that `ClassVar` is not a class, and you can't use it with `isinstance()` or `issubclass()`. It does not change Python runtime behavior – it's only for type checkers such as mypy (and also helpful for human readers).

You can also omit the square brackets and the variable type in a `ClassVar` annotation, but this might not do what you'd expect:

```
class A:
    y: ClassVar = 0  # Type implicitly Any!
```

In this case the type of the attribute will be implicitly `Any`. This behavior will change in the future, since it's surprising.

Note: A `ClassVar` type parameter cannot include type variables: `ClassVar[T]` and `ClassVar[List[T]]` are both invalid if `T` is a type variable (see *Defining generic classes* for more about type variables).

9.4 Overriding statically typed methods

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

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

class Derived1(Base):
    def f(self, x: str) -> None:    # Error: type of 'x' incompatible
        ...

class Derived2(Base):
```

(continues on next page)

(continued from previous page)

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

class Derived3(Base):
    def f(self, x: int) -> None:    # OK
        ...

class Derived4(Base):
    def f(self, x: float) -> None:    # OK: mypy treats int as a subtype of float
        ...

class Derived5(Base):
    def f(self, x: int, y: int = 0) -> None:    # OK: accepts more than the base
        ...                                     #     class method
```

Note: You can also vary return types **covariantly** in overriding. For example, you could override the return type `Iterable[int]` with a subtype such as `List[int]`. Similarly, you can vary argument types **contravariantly** – subclasses can have more general argument types.

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.

As always, relying on dynamically typed code can be unsafe. 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 Base:
    def inc(self, x: int) -> int:
        return x + 1

class Derived(Base):
    def inc(self, x):    # Override, dynamically typed
        return 'hello'  # Incompatible with 'Base', but no mypy error
```

9.5 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 any *concrete* (non-abstract) subclass. You can define abstract base classes using the `abc.ABCMeta` metaclass and the `abc.abstractmethod` function decorator. Example:

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):
    @abstractmethod
    def eat(self, food: str) -> None: pass

    @property
    @abstractmethod
    def can_walk(self) -> bool: pass

class Cat(Animal):
    def eat(self, food: str) -> None:
        ...  # Body omitted
```

(continues on next page)

(continued from previous page)

```
    @property
    def can_walk(self) -> bool:
        return True

x = Animal()  # Error: 'Animal' is abstract due to 'eat' and 'can_walk'
y = Cat()     # OK
```

Note: In Python 2.7 you have to use `@abc.abstractproperty` to define an abstract property.

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.

Since you can't create instances of ABCs, they are most commonly used in type annotations. For example, this method accepts arbitrary iterables containing arbitrary animals (instances of concrete `Animal` subclasses):

```
def feed_all(animals: Iterable[Animal], food: str) -> None:
    for animal in animals:
        animal.eat(food)
```

There is one important peculiarity about how ABCs work in Python – whether a particular class is abstract or not is somewhat implicit. In the example below, `Derived` is treated as an abstract base class since `Derived` inherits an abstract `f` method from `Base` and doesn't explicitly implement it. The definition of `Derived` generates no errors from mypy, since it's a valid ABC:

```
from abc import ABCMeta, abstractmethod

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

class Derived(Base):  # No error -- Derived is implicitly abstract
    def g(self) -> None:
        ...
```

Attempting to create an instance of `Derived` will be rejected, however:

```
d = Derived()  # Error: 'Derived' is abstract
```

Note: It's a common error to forget to implement an abstract method. As shown above, the class definition will not generate an error in this case, but any attempt to construct an instance will be flagged as an error.

A class can inherit any number of classes, both abstract and concrete. As with normal overrides, a dynamically typed method can override or 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.

CHAPTER 10

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.

10.1 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):
```

(continues on next page)

(continued from previous page)

```
        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).

10.1.1 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]
```

10.1.2 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
```

10.1.3 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
```

10.1.4 Async protocols

These protocols can be useful in async code. See *Typing async/await* for more information.

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]
```

10.1.5 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]]
```

10.2 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:
        ...  # Empty method body (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 `SupportsClose` 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.

10.3 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 NotAProtocol(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: NotAProtocol = 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.

10.4 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

    @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: TreeLike = SimpleTree(0)  # OK
```

10.5 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.

10.6 Callback protocols

Protocols can be used to define flexible callback types that are hard (or even impossible) to express using the `Callable[...]` syntax, such as variadic, overloaded, and complex generic callbacks. They are defined with a special `__call__` member:

```
from typing import Optional, Iterable, List
from typing_extensions import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: Optional[int] = None) -> List[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: Optional[int] = None) -> List[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: Optional[int]) -> List[bytes]:
    ...

batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has incompatible type because of
                         # different name and kind in the callback
```

Callback protocols and `Callable[...]` types can be used interchangeably. Keyword argument names in `__call__` methods must be identical, unless a double underscore prefix is used. For example:

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

T = TypeVar('T')

class Copy(Protocol):
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, __origin: T) -> T: ...

copy_a: Callable[[T], T]
copy_b: Copy

copy_a = copy_b  # OK
copy_b = copy_a  # Also OK
```

CHAPTER 11

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.)

11.1 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."""
        ...
```

11.2 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 12

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

12.1 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.

12.2 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 and type assertions*) 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 13

Casts and type assertions

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: object = [1]
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. 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.

Note: You can use an assertion if you want to perform an actual runtime check:

```
def foo(o: object) -> None:
    print(o + 5)  # Error: can't add 'object' and 'int'
    assert isinstance(o, int)
    print(o + 5)  # OK: type of 'o' is 'int' here
```

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.whatever()  # Type check error
y = cast(Any, x)
y.whatever()  # 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:

```
import math

def degrees_to_radians(degrees: 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

Stub files

Mypy uses stub files stored in the typeshed repository to determine the types of standard library and third-party library functions, classes, and other definitions. You can also create your own stubs that will be used to type check your code. The basic properties of stubs were introduced back in *Library stubs and typeshed*.

15.1 Creating a stub

Here is an overview of how to create a stub file:

- Write a stub file for the library (or an arbitrary module) 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. Note that PEP 561 stub-only packages must be installed, and may not be pointed at through the `MYPYPATH` (see *PEP 561 support*).

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.

15.2 Stub file syntax

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 16

Generics

This section explains how you can define your own generic classes that take one or more type parameters, similar to built-in types such as `List[X]`. User-defined generics are a moderately advanced feature and you can get far without ever using them – feel free to skip this section and come back later.

16.1 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: 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)
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
```

16.2 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.

16.3 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
```

16.4 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]
```

A variable cannot have a type variable in its type unless the type variable is bound in a containing generic class or function.

16.5 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

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.

16.6 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
```

(continues on next page)

(continued from previous page)

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

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

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

16.7 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.

16.8 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.

16.9 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)
```

(continues on next page)

(continued from previous page)

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

# 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 and type assertions*.

16.10 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'
```

16.11 Generic type aliases

Type aliases can be generic. In this case they can be used in two ways: Subscripted aliases are equivalent to original types with substituted type variables, so the number of type arguments must match the number of free type variables in the 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)

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 just like other names. An alias can also target another alias, although building complex chains of aliases is not recommended – this impedes code readability, thus defeating the purpose of using aliases. Example:

```
from typing import TypeVar, Generic, Optional
from example1 import AliasType
from example2 import Vec

# AliasType and Vec are type aliases (Vec as defined above)

def fun() -> AliasType:
    ...

T = TypeVar('T')

class NewVec(Vec[T]):
    ...

for i, j in NewVec[int]():
    ...

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

Note: A type alias does not define a new 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.

CHAPTER 17

More types

This section introduces a few additional kinds of types, including `NoReturn`, `NewType`, `TypedDict`, and types for async code. It also discusses how to give functions more precise types using overloads. All of these are only situationally useful, so feel free to skip this section and come back when you have a need for some of them.

Here's a quick summary of what's covered here:

- `NoReturn` lets you tell mypy that a function never returns normally.
- `NewType` lets you define a variant of a type that is treated as a separate type by mypy but is identical to the original type at runtime. For example, you can have `UserId` as a variant of `int` that is just an `int` at runtime.
- `@overload` lets you define a function that can accept multiple distinct signatures. This is useful if you need to encode a relationship between the arguments and the return type that would be difficult to express normally.
- `TypedDict` lets you give precise types for dictionaries that represent objects with a fixed schema, such as `{'id': 1, 'items': ['x']}`.
- Async types let you type check programs using `async` and `await`.

17.1 The NoReturn type

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

```
from typing 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
```

In earlier Python versions you need to install `typing_extensions` using pip to use `NoReturn` in your code. Python 3 command line:

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

This works for Python 2:

```
pip install --upgrade typing-extensions
```

17.2 NewTypes

There are situations where you may want to avoid programming errors by creating simple derived classes that are only used to distinguish certain values from base class instances. 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
```

(continues on next page)

(continued from previous page)

```
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
```

You cannot use `isinstance()` or `issubclass()` on the object returned by `NewType()`, because function objects don't support these operations. You cannot create subclasses of these objects either.

Note: 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
```

17.3 Function overloading

Sometimes the arguments and types in a function depend on each other in ways that can't be captured with a `Union`. For example, suppose we want to write a function that can accept x-y coordinates. If we pass in just a single x-y coordinate, we return a `ClickEvent` object. However, if we pass in two x-y coordinates, we return a `DragEvent` object.

Our first attempt at writing this function might look like this:

```
from typing import Union, Optional

def mouse_event(x1: int,
                y1: int,
                x2: Optional[int] = None,
                y2: Optional[int] = None) -> Union[ClickEvent, DragEvent]:
    if x2 is None and y2 is None:
        return ClickEvent(x1, y1)
    elif x2 is not None and y2 is not None:
        return DragEvent(x1, y1, x2, y2)
    else:
        raise TypeError("Bad arguments")
```

While this function signature works, it's too loose: it implies `mouse_event` could return either object regardless of the number of arguments we pass in. It also does not prohibit a caller from passing in the wrong number of ints: mypy would treat calls like `mouse_event(1, 2, 20)` as being valid, for example.

We can do better by using overloading which lets us give the same function multiple type annotations (signatures) to more accurately describe the function's behavior:

```
from typing import Union, overload

# Overload *variants* for 'mouse_event'.
# These variants give extra information to the type checker.
# They are ignored at runtime.

@overload
def mouse_event(x1: int, y1: int) -> ClickEvent: ...
@overload
def mouse_event(x1: int, y1: int, x2: int, y2: int) -> DragEvent: ...

# The actual *implementation* of 'mouse_event'.
# The implementation contains the actual runtime logic.
#
# It may or may not have type hints. If it does, mypy
# will check the body of the implementation against the
# type hints.
#
# Mypy will also check and make sure the signature is
# consistent with the provided variants.

def mouse_event(x1: int,
                y1: int,
                x2: Optional[int] = None,
                y2: Optional[int] = None) -> Union[ClickEvent, DragEvent]:
    if x2 is None and y2 is None:
        return ClickEvent(x1, y1)
    elif x2 is not None and y2 is not None:
```

(continues on next page)

(continued from previous page)

```
        return DragEvent(x1, y1, x2, y2)
    else:
        raise TypeError("Bad arguments")
```

This allows mypy to understand calls to `mouse_event` much more precisely. For example, mypy will understand that `mouse_event(5, 25)` will always have a return type of `ClickEvent` and will report errors for calls like `mouse_event(5, 25, 2)`.

As another example, suppose we want to write a custom container class that implements the `__getitem__` method (`[]` bracket indexing). If this method receives an integer we return a single item. If it receives a `slice`, we return a `Sequence` of items.

We can precisely encode this relationship between the argument and the return type by using overloads like so:

```
from typing import Sequence, TypeVar, Union, overload

T = TypeVar('T')

class MyList(Sequence[T]):
    @overload
    def __getitem__(self, index: int) -> T: ...

    @overload
    def __getitem__(self, index: slice) -> 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(...)
```

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

17.3.1 Runtime behavior

An overloaded function must consist of two or more overload *variants* followed by an *implementation*. The variants and the implementations must be adjacent in the code: think of them as one indivisible unit.

The variant bodies must all be empty; only the implementation is allowed to contain code. This is because at runtime, the variants are completely ignored: they're overridden by the final implementation function.

This means that an overloaded function is still an ordinary Python function! There is no automatic dispatch handling and you must manually handle the different types in the implementation (e.g. by using `if` statements and `isinstance` checks).

If you are adding an overload within a stub file, the implementation function should be omitted: stubs do not contain runtime logic.

Note: While we can leave the variant body empty using the `pass` keyword, the more common convention is to instead use the ellipsis (`...`) literal.

17.3.2 Type checking calls to overloads

When you call an overloaded function, mypy will infer the correct return type by picking the best matching variant, after taking into consideration both the argument types and arity. However, a call is never type checked against the implementation. This is why mypy will report calls like `mouse_event(5, 25, 3)` as being invalid even though it matches the implementation signature.

If there are multiple equally good matching variants, mypy will select the variant that was defined first. For example, consider the following program:

```
from typing import List, overload

@overload
def summarize(data: List[int]) -> float: ...

@overload
def summarize(data: List[str]) -> str: ...

def summarize(data):
    if not data:
        return 0.0
    elif isinstance(data[0], int):
        # Do int specific code
    else:
        # Do str-specific code

# What is the type of 'output'? float or str?
output = summarize([])
```

The `summarize([])` call matches both variants: an empty list could be either a `List[int]` or a `List[str]`. In this case, mypy will break the tie by picking the first matching variant: `output` will have an inferred type of `float`. The implementor is responsible for making sure `summarize` breaks ties in the same way at runtime.

However, there are two exceptions to the "pick the first match" rule. First, if multiple variants match due to an argument being of type `Any`, mypy will make the inferred type also be `Any`:

```
dynamic_var: Any = some_dynamic_function()

# output2 is of type 'Any'
output2 = summarize(dynamic_var)
```

Second, if multiple variants match due to one or more of the arguments being a union, mypy will make the inferred type be the union of the matching variant returns:

```
some_list: Union[List[int], List[str]]

# output3 is of type 'Union[float, str]'
output3 = summarize(some_list)
```

Note: Due to the "pick the first match" rule, changing the order of your overload variants can change how mypy type checks your program.

To minimize potential issues, we recommend that you:

1. Make sure your overload variants are listed in the same order as the runtime checks (e.g. `isinstance` checks) in your implementation.
2. Order your variants and runtime checks from most to least specific. (See the following section for an example).

17.3.3 Type checking the variants

Mypy will perform several checks on your overload variant definitions to ensure they behave as expected. First, mypy will check and make sure that no overload variant is shadowing a subsequent one. For example, consider the following function which adds together two `Expression` objects, and contains a special-case to handle receiving two `Literal` types:

```
from typing import overload, Union

class Expression:
    # ...snip...

class Literal(Expression):
    # ...snip...

# Warning -- the first overload variant shadows the second!

@overload
def add(left: Expression, right: Expression) -> Expression: ...

@overload
def add(left: Literal, right: Literal) -> Literal: ...

def add(left: Expression, right: Expression) -> Expression:
    # ...snip...
```

While this code snippet is technically type-safe, it does contain an anti-pattern: the second variant will never be selected! If we try calling `add(Literal(3), Literal(4))`, mypy will always pick the first variant and evaluate the function call to be of type `Expression`, not `Literal`. This is because `Literal` is a subtype of `Expression`, which means the "pick the first match" rule will always halt after considering the first overload.

Because having an overload variant that can never be matched is almost certainly a mistake, mypy will report an error. To fix the error, we can either 1) delete the second overload or 2) swap the order of the overloads:

```
# Everything is ok now -- the variants are correctly ordered
# from most to least specific.

@overload
def add(left: Literal, right: Literal) -> Literal: ...

@overload
def add(left: Expression, right: Expression) -> Expression: ...

def add(left: Expression, right: Expression) -> Expression:
    # ...snip...
```

Mypy will also type check the different variants and flag any overloads that have inherently unsafely overlapping variants. For example, consider the following unsafe overload definition:

```
from typing import overload, Union

@overload
def unsafe_func(x: int) -> int: ...

```

(continues on next page)

(continued from previous page)

```
@overload
def unsafe_func(x: object) -> str: ...

def unsafe_func(x: object) -> Union[int, str]:
    if isinstance(x, int):
        return 42
    else:
        return "some string"
```

On the surface, this function definition appears to be fine. However, it will result in a discrepancy between the inferred type and the actual runtime type when we try using it like so:

```
some_obj: object = 42
unsafe_func(some_obj) + " danger danger"  # Type checks, yet crashes at runtime!
```

Since `some_obj` is of type `object`, mypy will decide that `unsafe_func` must return something of type `str` and concludes the above will type check. But in reality, `unsafe_func` will return an int, causing the code to crash at runtime!

To prevent these kinds of issues, mypy will detect and prohibit inherently unsafely overlapping overloads on a best-effort basis. Two variants are considered unsafely overlapping when both of the following are true:

1. All of the arguments of the first variant are compatible with the second.
2. The return type of the first variant is *not* compatible with (e.g. is not a subtype of) the second.

So in this example, the `int` argument in the first variant is a subtype of the `object` argument in the second, yet the `int` return type not is a subtype of `str`. Both conditions are true, so mypy will correctly flag `unsafe_func` as being unsafe.

However, mypy will not detect *all* unsafe uses of overloads. For example, suppose we modify the above snippet so it calls `summarize` instead of `unsafe_func`:

```
some_list: List[str] = []
summarize(some_list) + "danger danger"  # Type safe, yet crashes at runtime!
```

We run into a similar issue here. This program type checks if we look just at the annotations on the overloads. But since `summarize(...)` is designed to be biased towards returning a float when it receives an empty list, this program will actually crash during runtime.

The reason mypy does not flag definitions like `summarize` as being potentially unsafe is because if it did, it would be extremely difficult to write a safe overload. For example, suppose we define an overload with two variants that accept types `A` and `B` respectively. Even if those two types were completely unrelated, the user could still potentially trigger a runtime error similar to the ones above by passing in a value of some third type `C` that inherits from both `A` and `B`.

Thankfully, these types of situations are relatively rare. What this does mean, however, is that you should exercise caution when designing or using an overloaded function that can potentially receive values that are an instance of two seemingly unrelated types.

17.3.4 Type checking the implementation

The body of an implementation is type-checked against the type hints provided on the implementation. For example, in the `MyList` example up above, the code in the body is checked with argument list `index: Union[int, slice]` and a return type of `Union[T, Sequence[T]]`. If there are no annotations on the implementation, then the body is not type checked. If you want to force mypy to check the body anyways, use the `--check-untyped-defs` flag (*more details here*).

The variants must also also be compatible with the implementation type hints. In the `MyList` example, mypy will check that the parameter type `int` and the return type `T` are compatible with `Union[int, slice]` and `Union[T, Sequence]` for the first variant. For the second variant it verifies the parameter type `slice` and the return type `Sequence[T]` are compatible with `Union[int, slice]` and `Union[T, Sequence]`.

Note: The overload semantics documented above are new as of mypy 0.620.

Previously, mypy used to perform type erasure on all overload variants. For example, the `summarize` example from the previous section used to be illegal because `List[str]` and `List[int]` both erased to just `List[Any]`. This restriction was removed in mypy 0.620.

Mypy also previously used to select the best matching variant using a different algorithm. If this algorithm failed to find a match, it would default to returning `Any`. The new algorithm uses the "pick the first match" rule and will fall back to returning `Any` only if the input arguments also contain `Any`.

17.4 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 `typing.Coroutine[Any, Any, T]`, which is a subtype of `Awaitable[T]`:

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

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

If you want to use coroutines in Python 3.4, which does 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:
        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):
```

(continues on next page)

(continued from previous page)

```
            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)

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.

17.5 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 a compatible with (a subtype of) a narrower TypedDict, assuming item types are compatible (*totality* also affects subtyping, as discussed below).

A TypedDict object is not a subtype of the regular `Dict[...]` type (and vice versa), since `Dict` allows arbitrary keys to be added and removed, unlike TypedDict. However, any TypedDict object is a subtype of (that is, compatible with) `Mapping[str, object]`, since `typing.Mapping` only provides read-only access to the dictionary items:

```
def print_typed_dict(obj: Mapping[str, object]) -> None:
    for key, value in obj.items():
```

(continues on next page)

(continued from previous page)

```
        print('{}: {}'.format(key, value))

print_typed_dict(Movie(name='Toy Story', year=1995))  # OK
```

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
```

17.5.1 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 TypedDict is not valid when a partial one is expected.

17.5.2 Supported operations

TypedDict objects support a subset of dictionary operations and methods. You must use string literals as keys when calling most of the methods, as otherwise mypy won't be able to check that the key is valid. List of supported operations:

- Anything included in `typing.Mapping`:
  - `d[key]`
  - `key in d`

    - `len(d)`
    - `for key in d` (iteration)
    - `d.get(key[, default])`
    - `d.keys()`
    - `d.values()`
    - `d.items()`
- `d.copy()`
- `d.setdefault(key, default)`
- `d1.update(d2)`
- `d.pop(key[, default])` (partial TypedDicts only)
- `del d[key]` (partial TypedDicts only)

In Python 2 code, these methods are also supported:

- `has_key(key)`
- `viewitems()`
- `viewkeys()`
- `viewvalues()`

Note: `clear()` and `popitem()` are not supported since they are unsafe – they could delete required TypedDict items that are not visible to mypy because of structural subtyping.

17.5.3 Class-based syntax

An alternative, class-based syntax to define a TypedDict is supported in Python 3.6 and later:

```
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`.

17.5.4 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 18

Literal types

Note: Literal is an officially supported feature, but is highly experimental and should be considered to be in alpha stage. It is very likely that future releases of mypy will modify the behavior of literal types, either by adding new features or by tuning or removing problematic ones.

Literal types let you indicate that an expression is equal to some specific primitive value. For example, if we annotate a variable with type `Literal["foo"]`, mypy will understand that variable is not only of type `str`, but is also equal to specifically the string `"foo"`.

This feature is primarily useful when annotating functions that behave differently based on the exact value the caller provides. For example, suppose we have a function `fetch_data(...)` that returns `bytes` if the first argument is `True`, and `str` if it's `False`. We can construct a precise type signature for this function using `Literal[...]` and overloads:

```
from typing import overload, Union
from typing_extensions import Literal

# The first two overloads use Literal[...] so we can
# have precise return types:

@overload
def fetch_data(raw: Literal[True]) -> bytes: ...
@overload
def fetch_data(raw: Literal[False]) -> str: ...

# The last overload is a fallback in case the caller
# provides a regular bool:

@overload
def fetch_data(raw: bool) -> Union[bytes, str]: ...

def fetch_data(raw: bool) -> Union[bytes, str]:
    # Implementation is omitted
```

(continues on next page)

(continued from previous page)

```
    ...

reveal_type(fetch_data(True))        # Revealed type is 'bytes'
reveal_type(fetch_data(False))       # Revealed type is 'str'

# Variables declared without annotations will continue to have an
# inferred type of 'bool'.

variable = True
reveal_type(fetch_data(variable))    # Revealed type is 'Union[bytes, str]'
```

18.1 Parameterizing Literals

Literal types may contain one or more literal bools, ints, strs, and bytes. However, literal types **cannot** contain arbitrary expressions: types like `Literal[my_string.trim()]`, `Literal[x > 3]`, or `Literal[3j + 4]` are all illegal.

Literals containing two or more values are equivalent to the union of those values. So, `Literal[-3, b"foo", True]` is equivalent to `Union[Literal[-3], Literal[b"foo"], Literal[True]]`. This makes writing more complex types involving literals a little more convenient.

Literal types may also contain `None`. Mypy will treat `Literal[None]` as being equivalent to just `None`. This means that `Literal[4, None]`, `Union[Literal[4], None]`, and `Optional[Literal[4]]` are all equivalent.

Literals may also contain aliases to other literal types. For example, the following program is legal:

```
PrimaryColors = Literal["red", "blue", "yellow"]
SecondaryColors = Literal["purple", "green", "orange"]
AllowedColors = Literal[PrimaryColors, SecondaryColors]

def paint(color: AllowedColors) -> None: ...

paint("red")        # Type checks!
paint("turquoise")  # Does not type check
```

Literals may not contain any other kind of type or expression. This means doing `Literal[my_instance]`, `Literal[Any]`, `Literal[3.14]`, or `Literal[{"foo": 2, "bar": 5}]` are all illegal.

Future versions of mypy may relax some of these restrictions. For example, we plan on adding support for using enum values inside `Literal[...]` in an upcoming release.

18.2 Declaring literal variables

You must explicitly add an annotation to a variable to declare that it has a literal type:

```
a: Literal[19] = 19
reveal_type(a)          # Revealed type is 'Literal[19]'
```

In order to preserve backwards-compatibility, variables without this annotation are **not** assumed to be literals:

```
b = 19
reveal_type(b)          # Revealed type is 'int'
```

If you find repeating the value of the variable in the type hint to be tedious, you can instead change the variable to be *Final*:

```
from typing_extensions import Final, Literal

def expects_literal(x: Literal[19]) -> None: pass

c: Final = 19

reveal_type(c)           # Revealed type is 'int'
expects_literal(c)       # ...but this type checks!
```

If you do not provide an explicit type in the Final, the type of `c` becomes context-sensitive: mypy will basically try "substituting" the original assigned value whenever it's used before performing type checking. So, mypy will type-check the above program almost as if it were written like so:

```
from typing_extensions import Final, Literal

def expects_literal(x: Literal[19]) -> None: pass

reveal_type(19)
expects_literal(19)
```

This is why `expects_literal(19)` type-checks despite the fact that `reveal_type(c)` reports `int`.

So while changing a variable to be Final is not quite the same thing as adding an explicit `Literal[...]` annotation, it often leads to the same effect in practice.

18.3 Limitations

Mypy will not understand expressions that use variables of type `Literal[..]` on a deep level. For example, if you have a variable `a` of type `Literal[3]` and another variable `b` of type `Literal[5]`, mypy will infer that `a + b` has type `int`, **not** type `Literal[8]`.

The basic rule is that literal types are treated as just regular subtypes of whatever type the parameter has. For example, `Literal[3]` is treated as a subtype of `int` and so will inherit all of `int`'s methods directly. This means that `Literal[3].__add__` accepts the same arguments and has the same return type as `int.__add__`.

CHAPTER 19

Final names, methods and classes

This section introduces these related features:

1. *Final names* are variables or attributes that should not reassigned after initialization. They are useful for declaring constants.
2. *Final methods* should not be overridden in a subclass.
3. *Final classes* should not be subclassed.

All of these are only enforced by mypy, and only in annotated code. They is no runtime enforcement by the Python runtime.

Note: These are experimental features. They might change in later versions of mypy. The *final* qualifiers are available through the `typing_extensions` package on PyPI.

19.1 Final names

You can use the `typing_extensions.Final` qualifier to indicate that a name or attribute should not be reassigned, redefined, or overridden. This is often useful for module and class level constants as a way to prevent unintended modification. Mypy will prevent further assignments to final names in type-checked code:

```
from typing_extensions import Final

RATE: Final = 3000

class Base:
    DEFAULT_ID: Final = 0

RATE = 300  # Error: can't assign to final attribute
Base.DEFAULT_ID = 1  # Error: can't override a final attribute
```

Another use case for final attributes is to protect certain attributes from being overridden in a subclass:

```
from typing_extensions import Final

class Window:
    BORDER_WIDTH: Final = 2.5
    ...

class ListView(Window):
    BORDER_WIDTH = 3  # Error: can't override a final attribute
```

You can use `@property` to make an attribute read-only, but unlike `Final`, it doesn't work with module attributes, and it doesn't prevent overriding in subclasses.

19.1.1 Syntax variants

You can use `Final` in one of these forms:

- You can provide an explicit type using the syntax `Final[<type>]`. Example:

  ```
  ID: Final[float] = 1
  ```

- You can omit the type:

  ```
  ID: Final = 1
  ```

  Here mypy will infer type `int` for `ID`. Note that unlike for generic classes this is *not* the same as `Final[Any]`.

- In class bodies and stub files you can omit the right hand side and just write `ID: Final[float]`.
- Finally, you can write `self.id: Final = 1` (also optionally with a type in square brackets). This is allowed *only* in `__init__` methods, so that the final instance attribute is assigned only once when an instance is created.

19.1.2 Details of using Final

These are the two main rules for defining a final name:

- There can be *at most one* final declaration per module or class for a given attribute. There can't be separate class-level and instance-level constants with the same name.
- There must be *exactly one* assignment to a final name.

A final attribute declared in a class body without an initializer must be initialized in the `__init__` method (you can skip the initializer in stub files):

```
class ImmutablePoint:
    x: Final[int]
    y: Final[int]  # Error: final attribute without an initializer

    def __init__(self) -> None:
        self.x = 1  # Good
```

`Final` can only be used as the outermost type in assignments or variable annotations. Using it in any other position is an error. In particular, `Final` can't be used in annotations for function arguments:

```
x: List[Final[int]] = []  # Error!

def fun(x: Final[List[int]]) ->  None:  # Error!
    ...
```

`Final` and `ClassVar` should not be used together. Mypy will infer the scope of a final declaration automatically depending on whether it was initialized in the class body or in `__init__`.

A final attribute can't be overridden by a subclass (even with another explicit final declaration). Note however that a final attribute can override a read-only property:

```
class Base:
    @property
    def ID(self) -> int: ...

class Derived(Base):
    ID: Final = 1  # OK
```

Declaring a name as final only guarantees that the name wll not be re-bound to another value. It doesn't make the value immutable. You can use immutable ABCs and containers to prevent mutating such values:

```
x: Final = ['a', 'b']
x.append('c')  # OK

y: Final[Sequence[str]] = ['a', 'b']
y.append('x')  # Error: Sequence is immutable
z: Final = ('a', 'b')  # Also an option
```

19.2 Final methods

Like with attributes, sometimes it is useful to protect a method from overriding. You can use the `typing_extensions.final` decorator for this purpose:

```
from typing_extensions import final

class Base:
    @final
    def common_name(self) -> None:
        ...

class Derived(Base):
    def common_name(self) -> None:  # Error: cannot override a final method
        ...
```

This `@final` decorator can be used with instance methods, class methods, static methods, and properties.

For overloaded methods you should add `@final` on the implementation to make it final (or on the first overload in stubs):

```
from typing import Any, overload

class Base:
    @overload
    def method(self) -> None: ...
    @overload
```

(continues on next page)

(continued from previous page)

```
    def method(self, arg: int) -> int: ...
    @final
    def method(self, x=None):
        ...
```

19.3 Final classes

You can apply the `typing_extensions.final` decorator to a class to indicate to mypy that it should not be subclassed:

```
from typing_extensions import final

@final
class Leaf:
    ...

class MyLeaf(Leaf):  # Error: Leaf can't be subclassed
    ...
```

The decorator acts as a declaration for mypy (and as documentation for humans), but it doesn't actually prevent subclassing at runtime.

Here are some situations where using a final class may be useful:

- A class wasn't designed to be subclassed. Perhaps subclassing would not work as expected, or subclassing would be error-prone.
- Subclassing would make code harder to understand or maintain. For example, you may want to prevent unnecessarily tight coupling between base classes and subclasses.
- You want to retain the freedom to arbitrarily change the class implementation in the future, and these changes might break subclasses.

CHAPTER 20

Metaclasses

A metaclass is a class that describes the construction and behavior of other classes, similarly to how classes describe the construction and behavior of objects. The default metaclass is `type`, but it's possible to use other metaclasses. Metaclasses allows one to create "a different kind of class", such as Enums, NamedTuples and singletons.

Mypy has some special understanding of `ABCMeta` and `EnumMeta`.

20.1 Defining a metaclass

```
class M(type):
    pass

class A(metaclass=M):
    pass
```

In Python 2, the syntax for defining a metaclass is different:

```
class A(object):
    __metaclass__ = M
```

Mypy also supports using the six library to define metaclass in a portable way:

```
import six

class A(six.with_metaclass(M)):
    pass

@six.add_metaclass(M)
class C(object):
    pass
```

20.2 Metaclass usage example

Mypy supports the lookup of attributes in the metaclass:

```
from typing import Type, TypeVar, ClassVar
T = TypeVar('T')

class M(type):
    count: ClassVar[int] = 0

    def make(cls: Type[T]) -> T:
        M.count += 1
        return cls()

class A(metaclass=M):
    pass

a: A = A.make()  # make() is looked up at M; the result is an object of type A
print(A.count)

class B(A):
    pass

b: B = B.make()  # metaclasses are inherited
print(B.count + " objects were created")  # Error: Unsupported operand types for + (
↪"int" and "str")
```

20.3 Gotchas and limitations of metaclass support

Note that metaclasses pose some requirements on the inheritance structure, so it's better not to combine metaclasses and class hierarchies:

```
class M1(type): pass
class M2(type): pass

class A1(metaclass=M1): pass
class A2(metaclass=M2): pass

class B1(A1, metaclass=M2): pass  # Mypy Error: Inconsistent metaclass structure for
↪'B1'
# At runtime the above definition raises an exception
# TypeError: metaclass conflict: the metaclass of a derived class must be a (non-
↪strict) subclass of the metaclasses of all its bases

# Same runtime error as in B1, but mypy does not catch it yet
class B12(A1, A2): pass
```

- Mypy does not understand dynamically-computed metaclasses, such as `class A(metaclass=f()): ...`
- Mypy does not and cannot understand arbitrary metaclass code.
- Mypy only recognizes subclasses of `type` as potential metaclasses.

CHAPTER 21

Running mypy and managing imports

The *Getting started* page should have already introduced you to the basics of how to run mypy – pass in the files and directories you want to type check via the command line:

```
$ mypy foo.py bar.py some_directory
```

This page discusses in more detail how exactly to specify what files you want mypy to type check, how mypy discovers imported modules, and recommendations on how to handle any issues you may encounter along the way.

If you are interested in learning about how to configure the actual way mypy type checks your code, see our *The mypy command line* guide.

21.1 Specifying code to be checked

Mypy lets you specify what files it should type check in several different ways.

1. First, you can pass in paths to Python files and directories you want to type check. For example:

   ```
   $ mypy file_1.py foo/file_2.py file_3.pyi some/directory
   ```

   The above command tells mypy it should type check all of the provided files together. In addition, mypy will recursively type check the entire contents of any provided directories.

   For more details about how exactly this is done, see *Mapping file paths to modules*.

2. Second, you can use the `-m` flag (long form: `--module`) to specify a module name to be type checked. The name of a module is identical to the name you would use to import that module within a Python program. For example, running:

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

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

   Mypy will use an algorithm very similar to the one Python uses to find where modules and imports are located on the file system. For more details, see *How imports are found*.

3. Third, you can use the `-p` (long form: `--package`) flag to specify a package to be (recursively) type checked. This flag is almost identical to the `-m` flag except that if you give it a package name, mypy will recursively type check all submodules and subpackages of that package. For example, running:

```
$ mypy -p html
```

   ... will type check the entire `html` package (of library stubs). In contrast, if we had used the `-m` flag, mypy would have type checked just `html`'s `__init__.py` file and anything imported from there.

   Note that we can specify multiple packages and modules on the command line. For example:

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

4. Fourth, you can also instruct mypy to directly type check small strings as programs by using the `-c` (long form: `--command`) flag. For example:

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

   ... will type check the above string as a mini-program (and in this case, will report that `List[int]` is not callable).

21.2 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.txt)
```

you can use this instead:

```
$ mypy @file_of_files.txt
```

This file can technically also contain any command line flag, not just file paths. However, if you want to configure many different flags, the recommended approach is to use a *configuration file* instead.

21.3 How mypy handles imports

When mypy encounters an `import` statement, it will first *attempt to locate* that module or type stubs for that module in the file system. Mypy will then type check the imported module. There are three different outcomes of this process:

1. Mypy is unable to follow the import: the module either does not exist, or is a third party library that does not use type hints.
2. Mypy is able to follow and type check the import, but you did not want mypy to type check that module at all.
3. Mypy is able to successfully both follow and type check the module, and you want mypy to type check that module.

The third outcome is what mypy will do in the ideal case. The following sections will discuss what to do in the other two cases.

21.3.1 Missing imports

When you import a module, mypy may report that it is unable to follow the import.

This can cause a lot of errors that look 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 'this_module_does_not_exist'
```

There are several different things you can try doing, depending on the exact nature of the module.

If the module is a part of your own codebase, try:

1. Making sure your import does not contain a typo.
2. Reading the *How imports are found* section below to make sure you understand how exactly mypy searches for and finds modules and modify how you're invoking mypy accordingly.
3. Adding the directory containing that module to either the `MYPYPATH` environment variable or the `mypy_path` *config file option*.

   Note: if the module you are trying to import is actually a *submodule* of some package, you should add the directory containing the *entire* package to `MYPYPATH`. For example, suppose you are trying to add the module `foo.bar.baz`, which is located at `~/foo-project/src/foo/bar/baz.py`. In this case, you should add `~/foo-project/src` to `MYPYPATH`.

If the module is a third party library, you must make sure that there are type hints available for that library. Mypy by default will not attempt to infer the types of any 3rd party libraries you may have installed unless they either have declared themselves to be *PEP 561 compliant stub package* or have registered themselves on typeshed, the repository of types for the standard library and some 3rd party libraries.

If you are getting an import-related error, this means the library you are trying to use has done neither of these things. In that case, you can try:

1. Searching to see if there is a *PEP 561 compliant stub package*. corresponding to your third party library. Stub packages let you install type hints independently from the library itself.
2. *Writing your own stub files* containing type hints for the library. You can point mypy at your type hints either by passing them in via the command line, by adding the location to the *MYPYPATH* environment variable, or by using the `mypy_path` *config file option*.

   Note that if you decide to write your own stub files, they don't need to be complete! A good strategy is to add stubs for just the parts of the library you need and iterate on them over time.

   If you want to share your work, you can try contributing your stubs back to the library – see our documentation on creating *PEP 561 compliant packages*.

If the module is a third party library, but you cannot find any existing type hints nor have time to write your own, you can *silence* the errors:

1. To silence a *single* missing import error, add a *# type: ignore* at the end of the line containing the import.
2. To silence *all* missing import imports errors from a single library, add a section to your *mypy config file* for that library setting `ignore_missing_imports` to True. For example, suppose your codebase makes heavy use of an (untyped) library named `foobar`. You can silence all import errors associated with that library and that library alone by adding the following section to your config file:

   ```
   [mypy-foobar]
   ignore_missing_imports = True
   ```

Note: this option is equivalent to adding a `# type: ignore` to every import of `foobar` in your codebase. For more information, see the documentation about configuring *import discovery* in config files.

3. To silence *all* missing import errors for *all* libraries in your codebase, invoke mypy with the `--ignore-missing-imports` command line flag or set the `ignore_missing_imports` *config file option* to True in the *global* section of your mypy config file:

```
[mypy]
ignore_missing_imports = True
```

We recommend using this approach only as a last resort: it's equivalent to adding a `# type: ignore` to all unresolved imports in your codebase.

If the module is a part of the standard library, try:

1. Updating mypy and re-running it. It's possible type hints for that corner of the standard library were added in a later version of mypy.
2. Filing a bug report on typeshed, the repository of type hints for the standard library that comes bundled with mypy. You can expedite this process by also submitting a pull request fixing the bug.

   Changes to typeshed will come bundled with mypy the next time it's released. In the meantime, you can add a `# type: ignore` to silence any relevant errors. After upgrading, we recommend running mypy using the `--warn-unused-ignores` flag to help you find any `# type: ignore` annotations you no longer need.

21.3.2 Following imports

Mypy is designed to *doggedly follow all imports*, even if the imported module is not a file you explicitly wanted mypy to check.

For example, suppose we have two modules `mycode.foo` and `mycode.bar`: the former has type hints and the latter does not. We run `mypy -m mycode.foo` and mypy discovers that `mycode.foo` imports `mycode.bar`.

How do we want mypy to type check `mycode.bar`? We can configure the desired behavior by using the `--follow-imports` flag. This flag accepts one of four string values:

- `normal` (the default) follows all imports normally and type checks all top level code (as well as the bodies of all functions and methods with at least one type annotation in the signature).
- `silent` behaves in the same way as `normal` but will additionally *suppress* any error messages.
- `skip` will *not* follow imports and instead will silently replace the module (and *anything imported from it*) with an object of type `Any`.
- `error` behaves in the same way as `skip` but is not quite as silent – it will flag the import as an error, like this:

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

If you are starting a new codebase and plan on using type hints from the start, we recommend you use either `--follow-imports=normal` (the default) or `--follow-imports=error`. Either option will help make sure you are not skipping checking any part of your codebase by accident.

If you are planning on adding type hints to a large, existing code base, we recommend you start by trying to make your entire codebase (including files that do not use type hints) pass under `--follow-imports=normal`. This is usually not too difficult to do: mypy is designed to report as few error messages as possible when it is looking at unannotated code.

If doing this is intractable, we recommend passing mypy just the files you want to type check and use `--follow-imports=silent`. Even if mypy is unable to perfectly type check a file, it can still glean some useful information by parsing it (for example, understanding what methods a given object has). See *Using mypy with an existing codebase* for more recommendations.

We do not recommend using `skip` unless you know what you are doing: while this option can be quite powerful, it can also cause many hard-to-debug errors.

21.4 Mapping file paths to modules

One of the main ways you can tell mypy what files to type check is by providing mypy the paths to those files. For example:

```
$ mypy file_1.py foo/file_2.py file_3.pyi some/directory
```

This section describes how exactly mypy maps the provided paths to modules to type check.

- 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 `__main__` (matching the behavior of the Python interpreter), unless `--scripts-are-modules` is passed.

21.5 How imports are found

When mypy encounters an `import` statement or receives module names from the command line via the `--module` or `--package` flags, mypy 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 `mypy_path` *config file option*.

- The directories containing the sources given on the command line (see below).
- The installed packages marked as safe for type checking (see *PEP 561 support*)
- The relevant directories of the typeshed repo.

Note: You cannot point to a PEP 561 package via the MYPYPATH, it must be installed (see *PEP 561 support*)

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 for 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.)

CHAPTER 22

The mypy command line

This section documents mypy's command line interface. You can view a quick summary of the available flags by running `mypy --help`.

Note: Command line flags are liable to change between releases.

22.1 Specifying what to type check

By default, you can specify what code you want mypy to type check by passing in the paths to what you want to have type checked:

```
$ mypy foo.py bar.py some_directory
```

Note that directories are checked recursively.

Mypy also lets you specify what code to type check in several other ways. A short summary of the relevant flags is included below: for full details, see *Running mypy and managing imports*.

`-m MODULE`, `--module MODULE` Asks mypy to type check the provided module. This flag may be repeated multiple times.

Mypy *will not* recursively type check any submodules of the provided module.

`-p PACKAGE`, `--package PACKAGE` Asks mypy to type check the provided package. This flag may be repeated multiple times.

Mypy *will* recursively type check any submodules of the provided package. This flag is identical to `--module` apart from this behavior.

`-c PROGRAM_TEXT`, `--command PROGRAM_TEXT` Asks mypy to type check the provided string as a program.

22.2 Config file

--config-file CONFIG_FILE This flag makes mypy read configuration settings 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's 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.

--warn-unused-configs This flag makes mypy warn about unused `[mypy-<pattern>]` config file sections.

22.3 Import discovery

The following flags customize how exactly mypy discovers and follows imports.

--namespace-packages This flag enables import discovery to use namespace packages (see PEP 420). In particular, this allows discovery of imported packages that don't have an `__init__.py` (or `__init__.pyi`) file.

Namespace packages are found (using the PEP 420 rules, which prefers "classic" packages over namespace packages) along the module search path – this is primarily set from the source files passed on the command line, the `MYPYPATH` environment variable, and the *mypy_path config option*.

Note that this only affects import discovery – for modules and packages explicitly passed on the command line, mypy still searches for `__init__.py[i]` files in order to determine the fully-qualified module/package name.

--ignore-missing-imports This flag makes mypy ignore all missing imports. It is equivalent to adding `# type: ignore` comments to all unresolved imports within your codebase.

Note that this flag does *not* suppress errors about missing names in successfully resolved modules. For example, if one has the following files:

```
package/__init__.py
package/mod.py
```

Then mypy will generate the following errors with `--ignore-missing-imports`:

```
import package.unknown       # No error, ignored
x = package.unknown.func()   # OK. 'func' is assumed to be of type 'Any'

from package import unknown           # No error, ignored
from package.mod import NonExisting  # Error: Module has no attribute 'NonExisting
↪'
```

For more details, see *Missing imports*.

--follow-imports {normal,silent,skip,error} This flag adjusts how mypy follows imported modules that were not explicitly passed in via the command line.

The default option is `normal`: mypy will follow and type check all modules. For more information on what the other options do, see *Following imports*.

--python-executable EXECUTABLE This flag 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.

`--no-site-packages` This flag 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`.

`--no-silence-site-packages` By default, mypy will suppress any error messages generated within PEP 561 compliant packages. Adding this flag will disable this behavior.

22.4 Platform configuration

By default, mypy will assume that you intend to run your code using the same operating system and Python version you are using to run mypy itself. The following flags let you modify this behavior.

For more information on how to use these flags, see *Python version and system platform checks*.

`--python-version X.Y` This flag will make mypy type check 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`.

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, use the `--no-site-packages` flag (see *Import discovery* for more details).

`-2, --py2` Equivalent to running `--python-version 2.7`.

`--platform PLATFORM` This flag will make mypy type check your code as if it were run under the given operating system. Without this option, mypy will default to using whatever operating system you are currently using.

The `PLATFORM` parameter may be any string supported by sys.platform.

`--always-true NAME` This flag will treat all variables named `NAME` as compile-time constants that are always true. This flag may be repeated.

`--always-false NAME` This flag will treat all variables named `NAME` as compile-time constants that are always false. This flag may be repeated.

22.5 Disallow dynamic typing

The `Any` type is used represent a value that has a *dynamic type*. The `--disallow-any` family of flags will disallow various uses of the `Any` type in a module – this lets us strategically disallow the use of dynamic typing in a controlled way.

The following options are available:

`--disallow-any-unimported` This flag 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` This flag 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 This flag disallows functions that have `Any` in their signature after decorator transformation.

--disallow-any-explicit This flag disallows explicit `Any` in type positions such as type annotations and generic type parameters.

--disallow-any-generics This flag 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]`).

--disallow-subclassing-any This flag 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.

22.6 Untyped definitions and calls

The following flags configure how mypy handles untyped function definitions or calls.

--disallow-untyped-calls This flag reports an error whenever a function with type annotations calls a function defined without annotations.

--disallow-untyped-defs This flag reports an error whenever it encounters a function definition without type annotations.

--disallow-incomplete-defs This flag reports an error whenever it encounters a partly annotated function definition.

--check-untyped-defs This flag is less severe than the previous two options – 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-untyped-decorators This flag reports an error whenever a function with type annotations is decorated with a decorator without annotations.

22.7 None and Optional handling

The following flags adjust how mypy handles values of type *None*. For more details, see *Disabling strict optional checking*.

--no-implicit-optional This flag causes mypy to stop treating arguments with a `None` default value as having an implicit `Optional[...]` type.

For example, by default mypy will assume that the `x` parameter is of type `Optional[int]` in the code snippet below since the default parameter is `None`:

```
def foo(x: int = None) -> None:
    print(x)
```

If this flag is set, the above snippet will no longer type check: we must now explicitly indicate that the type is `Optional[int]`:

```
def foo(x: Optional[int] = None) -> None:
    print(x)
```

--no-strict-optional This flag 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).

22.8 Configuring warnings

The follow flags enable warnings for code that is sound but is potentially problematic or redundant in some way.

--warn-redundant-casts This flag will make mypy report an error whenever your code uses an unnecessary cast that can safely be removed.

--warn-unused-ignores This flag will make mypy report an error whenever your code uses a `# type: ignore` comment on a line that is not actually generating an error message.

This flag, along with the `--warn-redundant-casts` flag, are both particularly useful when you are upgrading mypy. Previously, you may have needed to add casts or `# type: ignore` annotations to work around bugs in mypy or missing stubs for 3rd party libraries.

These two flags let you discover cases where either workarounds are no longer necessary.

--no-warn-no-return By default, mypy will generate errors when a function is missing return statements in some execution paths. The only exceptions are when:

- The function has a `None` or `Any` return type
- The function has an empty body or a body that is just ellipsis (`...`). Empty functions are often used for abstract methods.

Passing in `--no-warn-no-return` will disable these error messages in all cases.

--warn-return-any This flag causes mypy to generate a warning when returning a value with type `Any` from a function declared with a non- `Any` return type.

22.9 Miscellaneous strictness flags

This section documents any other flags that do not neatly fall under any of the above sections.

--allow-untyped-globals This flag causes mypy to suppress errors caused by not being able to fully infer the types of global and class variables.

--allow-redefinition By default, mypy won't allow a variable to be redefined with an unrelated type. This flag enables redefinion of a variable with an arbitrary type *in some contexts*: only redefinitions within the same block and nesting depth as the original definition are allowed. Example where this can be useful:

```
def process(items: List[str]) -> None:
    # 'items' has type List[str]
    items = [item.split() for item in items]
```

(continues on next page)

(continued from previous page)

```
    # 'items' now has type List[List[str]]
    ...
```

--strict-equality By default, mypy allows always-false comparisons like `42 == 'no'`. Use this flag to prohibit such comparisons of non-overlapping types, and similar identity and container checks:

```
from typing import Text

text: Text
if b'some bytes' in text:  # Error: non-overlapping check!
    ...
if text != b'other bytes':  # Error: non-overlapping check!
    ...

assert text is not None  # OK, this special case is allowed.
```

--strict This flag mode enables all optional error checking flags. You can see the list of flags enabled by strict mode in the full `mypy --help` output.

Note: the exact list of flags enabled by running `--strict` may change over time.

22.10 Configuring error messages

The following flags let you adjust how much detail mypy displays in error messages.

--show-error-context This flag will precede all errors with "note" messages explaining the context of the error. For example, consider the following program:

```
class Test:
    def foo(self, x: int) -> int:
        return x + "bar"
```

Mypy normally displays an error message that looks like this:

```
main.py:3: error: Unsupported operand types for + ("int" and "str")
```

If we enable this flag, the error message now looks like this:

```
main.py: note: In member "foo" of class "Test":
main.py:3: error: Unsupported operand types for + ("int" and "str")
```

--show-column-numbers This flag 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")
```

22.11 Incremental mode

By default, mypy will store type information into a cache. Mypy will use this information to avoid unnecessary recomputation when it type checks your code again. This can help speed up the type checking process, especially when most parts of your program have not changed since the previous mypy run.

If you want to speed up how long it takes to recheck your code beyond what incremental mode can offer, try running mypy in *daemon mode*.

--no-incremental This flag disables incremental mode: mypy will no longer reference the cache when re-run.

Note that mypy will still write out to the cache even when incremental mode is disabled: see the `--cache-dir` flag below for more details.

--cache-dir DIR By default, mypy stores all cache data inside of a folder named `.mypy_cache` in the current directory. This flag lets you change this folder. This flag can also be useful for controlling cache use when using *remote caching*.

Mypy will also always write to the cache even when incremental mode is disabled so it can "warm up" the cache. To disable writing to the cache, use `--cache-dir=/dev/null` (UNIX) or `--cache-dir=nul` (Windows).

--skip-version-check By default, mypy will ignore cache data generated by a different version of mypy. This flag disables that behavior.

22.12 Advanced flags

The following flags are useful mostly for people who are interested in developing or debugging mypy internals.

--pdb This flag will invoke the Python debugger when mypy encounters a fatal error.

--show-traceback, --tb If set, this flag will display a full traceback when mypy encounters a fatal error.

--custom-typing MODULE This flag lets you use a custom module as a substitute for the `typing` module.

--custom-typeshed-dir DIR This flag 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.

--warn-incomplete-stub This flag modifies both the `--disallow-untyped-defs` and `--disallow-incomplete-defs` flags so they also report errors if stubs in typeshed are missing type annotations or has incomplete annotations. If both flags are missing, `--warn-incomplete-stub` also does nothing.

This flag is mainly intended to be used by people who want contribute to typeshed and would like a convenient way to find gaps and omissions.

If you want mypy to report an error when your codebase *uses* an untyped function, whether that function is defined in typeshed or not, use the `--disallow-untyped-call` flag. See *Untyped definitions and calls* for more details.

--shadow-file SOURCE_FILE SHADOW_FILE When mypy is asked to type check `SOURCE_FILE`, this flag makes mypy read from and type check 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 type check the contents of `temp.py` instead of `original.py`, but error messages will still reference `original.py`.

22.13 Report generation

If these flags are set, mypy will generate a report in the specified format into the specified directory.

--any-exprs-report DIR Causes mypy to generate a text file report documenting how many expressions of type `Any` are present within your codebase.

--linecount-report DIR Causes mypy to generate a text file report documenting the functions and lines that are typed and untyped within your codebase.

--linecoverage-report DIR Causes mypy to generate a JSON file that maps each source file's absolute filename to a list of line numbers that belong to typed functions in that file.

--cobertura-xml-report DIR Causes mypy to generate a Cobertura XML type checking coverage report.

You must install the lxml library to generate this report.

--html-report DIR, --xslt-html-report DIR Causes mypy to generate an HTML type checking coverage report.

You must install the lxml library to generate this report.

--txt-report DIR, --xslt-txt-report DIR Causes mypy to generate a text file type checking coverage report.

You must install the lxml library to generate this report.

--junit-xml JUNIT_XML Causes mypy to generate a JUnit XML test result document with type checking results. This can make it easier to integrate mypy with continuous integration (CI) tools.

22.14 Miscellaneous

--find-occurrences CLASS.MEMBER This flag will make mypy print out all usages of a class member based on static type information. This feature is experimental.

--scripts-are-modules This flag 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 lets you check 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.)

CHAPTER 23

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, then `$XDG_CONFIG_HOME/mypy/config`, then `~/.config/mypy/config`, and finally `.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.

23.1 Config file format

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 fully-qualified module names, with some components optionally replaced by the '*' character (e.g. `foo.bar`, `foo.bar.*`, `foo.*.baz`). These sections specify additional flags that only apply to *modules* whose name matches at least one of the patterns.

  A pattern of the form `qualified_module_name` matches only the named module, while `qualified_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`).

  Patterns may also be "unstructured" wildcards, in which stars may appear in the middle of a name (e.g `site.*.migrations.*`). Stars match zero or more module components (so `site.*.migrations.*` can match `site.migrations`).

When options conflict, the precedence order for the configuration sections is:

1. Sections with concrete module names (`foo.bar`)
2. Sections with "unstructured" wildcard patterns (`foo.*.baz`), with sections later in the configuration file overriding sections earlier.
3. Sections with "well-structured" wildcard patterns (`foo.bar.*`), with more specific overriding more general.
4. Command line options.
5. Top-level configuration file options.

The difference in precedence order between "structured" patterns (by specificity) and "unstructured" patterns (by order in the file) is unfortunate, and is subject to change in future versions.

Note: The `warn_unused_configs` flag may be useful to debug misspelled section names.

Note: Configuration flags are liable to change between releases.

23.2 Examples

Here is an example of a `mypy.ini` file. To use this config file, place it at the root of your repo and run mypy.

```
# Global options:

[mypy]
python_version = 2.7
warn_return_any = True
warn_unused_configs = True

# Per-module options:

[mypy-mycode.foo.*]
disallow_untyped_defs = True

[mypy-mycode.bar]
warn_return_any = False

[mypy-somelibrary]
ignore_missing_imports = True
```

This config file specifies three global options in the `[mypy]` section. These three options will:

1. Type-check your entire project assuming it will be run using Python 2.7. (This is equivalent to using the `--python-version 2.7` or `--2` flag).
2. Report an error whenever a function returns a value that is inferred to have type `Any`.
3. Report any config options that are unused by mypy. (This will help us catch typos when making changes to our config file).

Next, this module specifies three per-module options. The first two options change how mypy type checks code in `mycode.foo.*` and `mycode.bar`, which we assume here are two modules that you wrote. The final config option

changes how mypy type checks `somelibrary`, which we assume here is some 3rd party library you've installed and are importing. These options will:

1. Selectively disallow untyped function definitions only within the `mycode.foo` package – that is, only for function definitions defined in the `mycode/foo` directory.
2. Selectively *disable* the "function is returning any" warnings within `mycode.bar` only. This overrides the global default we set earlier.
3. Suppress any error messages generated when your codebase tries importing the module `somelibrary`. This is useful if `somelibrary` is some 3rd party library missing type hints.

23.3 Per-module and global options

The following config options may be set either globally (in the `[mypy]` section) or on a per-module basis (in sections like `[mypy-foo.bar]`).

If you set an option both globally and for a specific module, the module configuration options take precedence. This lets you set global defaults and override them on a module-by-module basis. If multiple pattern sections match a module, *the options from the most specific section are used where they disagree*.

23.3.1 Import discovery

For more information, see the *import discovery* section of the command line docs.

Note: this section describes options that can be used both globally and per-module. See below for a list of import discovery options that may be used *only globally*.

`ignore_missing_imports` (bool, default False) Suppresses error messages about imports that cannot be resolved.

If this option is used in a per-module section, the module name should match the name of the *imported* module, not the module containing the import statement.

`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 provided 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.

If this option is used in a per-module section, the module name should match the name of the *imported* module, not the module containing the import statement.

`follow_imports_for_stubs` (bool, 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 conjunction with `follow_imports=error`, this can be used to make any use of a particular `typeshed` module an error.

23.3.2 Disallow dynamic typing

For more information, see the *disallowing dynamic typing* section of the command line docs.

`disallow_any_unimported` (bool, default False) Disallows usage of types that come from unfollowed imports (anything imported from an unfollowed import is automatically given a type of `Any`).

disallow_any_expr (bool, default False) Disallows all expressions in the module that have type `Any`.

disallow_any_decorated (bool, default False) Disallows functions that have `Any` in their signature after decorator transformation.

disallow_any_explicit (bool, default False) Disallows explicit `Any` in type positions such as type annotations and generic type parameters.

disallow_any_generics (bool, default False) Disallows usage of generic types that do not specify explicit type parameters.

disallow_subclassing_any (bool, default False) Disallows subclassing a value of type `Any`.

23.3.3 Untyped definitions and calls

For more information, see the *untyped definitions and calls* section of the command line docs.

disallow_untyped_calls (bool, default False) Disallows calling functions without type annotations from functions with type annotations.

disallow_untyped_defs (bool, default False) Disallows defining functions without type annotations or with incomplete type annotations.

disallow_incomplete_defs (bool, default False) Disallows defining functions with incomplete type annotations.

check_untyped_defs (bool, default False) Type-checks the interior of functions without type annotations.

disallow_untyped_decorators (bool, default False) Reports an error whenever a function with type annotations is decorated with a decorator without annotations.

23.3.4 None and optional handling

For more information, see the *None and optional handling* section of the command line docs.

no_implicit_optional (bool, default False) Changes the treatment of arguments with a default value of None by not implicitly making their type Optional.

strict_optional (bool, 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.

23.3.5 Configuring warnings

For more information, see the *configuring warnings* section of the command line docs.

warn_unused_ignores (bool, default False) Warns about unneeded `# type: ignore` comments.

warn_no_return (bool, default True) Shows errors for missing return statements on some execution paths.

warn_return_any (bool, default False) Shows a warning when returning a value with type `Any` from a function declared with a non- `Any` return type.

23.3.6 Suppressing errors

Note: these configuration options are available in the config file only. There is no analog available via the command line options.

show_none_errors (bool, default True) Shows errors related to strict `None` checking, if the global `strict_optional` flag is enabled.

ignore_errors (bool, default False) Ignores all non-fatal errors.

23.3.7 Miscellaneous strictness flags

allow_redefinition (bool, default False) Allows variables to be redefined with an arbitrary type, as long as the redefinition is in the same block and nesting level as the original definition.

strict_equality (bool, default False) Prohibit equality checks, identity checks, and container checks between non-overlapping types.

23.4 Global-only options

The following options may only be set in the global section (`[mypy]`).

23.4.1 Import discovery

For more information, see the *import discovery* section of the command line docs.

Note: this section describes only global-only import discovery options. See above for a list of import discovery options that may be used *both per-module and globally*.

namespace_packages (bool, default False) Enables PEP 420 style namespace packages. See *the corresponding flag* for more information.

python_executable (string) Specifies the path to the Python executable to inspect to collect a list of available *PEP 561 packages*. Defaults to the executable used to run mypy.

no_silence_site_packages (bool, default False) Enables reporting error messages generated within PEP 561 compliant packages. Those error messages are suppressed by default, since you are usually not able to control errors in 3rd party code.

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.

23.4.2 Platform configuration

For more information, see the *platform configuration* section of the command line docs.

python_version (string) Specifies the Python version used to parse and check the target program. The string should be in the format `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) Specifies a list of variables that mypy will treat as compile-time constants that are always true.

always_false (comma-separated list of strings) Specifies a list of variables that mypy will treat as compile-time constants that are always false.

23.4.3 Incremental mode

For more information, see the *incremental mode* section of the command line docs.

incremental (bool, default True) Enables *incremental mode*.

cache_dir (string, default .mypy_cache) Specifies the location where mypy stores incremental cache info. Note that the cache is only read when incremental mode is enabled but is always written to, unless the value is set to `/dev/nul` (UNIX) or `nul` (Windows).

skip_version_check (bool, default False) Makes mypy use incremental cache data even if it was generated by a different version of mypy. (By default, mypy will perform a version check and regenerate the cache if it was written by older versions of mypy.)

23.4.4 Configuring error messages

For more information, see the *configuring error messages* section of the command line docs.

show_error_context (bool, default False) Prefixes each error with the relevant context.

show_column_numbers (bool, default False) Shows column numbers in error messages.

23.4.5 Advanced options

For more information, see the *advanced flags* section of the command line docs.

pdb (bool, default False) Invokes pdb on fatal error.

show_traceback (bool, default False) Shows traceback on fatal error.

custom_typing_module (string) Specifies a custom module to use as a substitute for the `typing` module.

custom_typeshed_dir (string) Specifies an alternative directory to look for stubs instead of the default `typeshed` directory.

warn_incomplete_stub (bool, default False) Warns about missing type annotations in typeshed. This is only relevant in combination with `disallow_untyped_defs` or `disallow_incomplete_defs`.

23.4.6 Miscellaneous

warn_redundant_casts (bool, default False) Warns about casting an expression to its inferred type.

scripts_are_modules (bool, default False) Makes script `x` become module `x` instead of `__main__`. This is useful when checking multiple scripts in a single run.

warn_unused_configs (bool, default False) Warns about per-module sections in the config file that do not match any files processed when invoking mypy.

verbosity (integer, default 0) Controls how much debug output will be generated. Higher numbers are more verbose.

CHAPTER 24

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: 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.

24.1 Basic usage

The client utility `dmypy` is used to control the mypy daemon. Use `dmypy run -- <flags> <files>` to type-check a set of files (or directories). This will launch the daemon if it is not running. You can use almost arbitrary mypy flags after `--`. The daemon will always run on the current host. Example:

```
dmypy run -- --follow-imports=error prog.py pkg1/ pkg2/
```

Note: You'll need to use either the `--follow-imports=error` or the `--follow-imports=skip` option with dmypy because the current implementation can't follow imports. See *Following imports* for details on how these

work. You can also define these using a *configuration file*.

`dmypy run` will automatically restart the daemon if the configuration or mypy version changes.

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.

24.2 Additional features

While `dmypy run` is sufficient for most uses, some workflows (ones using *remote caching*, perhaps), require more precise control over the lifetime of the daemon process:

- `dmypy stop` stops the daemon.
- `dmypy start -- <flags>` starts the daemon but does not check any files. You can use almost arbitrary mypy flags after `--`.
- `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 run --timeout SECONDS -- <flags>` (or `start` or `restart`) to automatically shut down the daemon after inactivity. By default, the daemon runs until it's explicitly stopped.
- `dmypy check <files>` checks a set of files using an already running daemon.
- `dmypy status` checks whether a daemon is running. It prints a diagnostic and exits with `0` if there is a running daemon.

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.

24.3 Limitations

- You have to use either the `--follow-imports=error` or the `--follow-imports=skip` option because of an implementation limitation. This can be defined through the command line or through a *configuration file*.

CHAPTER 25

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.

25.1 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 and custom import hooks.

If you do not want to use typed packages, use the `--no-site-packages` flag to disable searching.

Note that stub-only packages (defined in PEP 561) cannot be used with `MYPYPATH`. If you want mypy to find the package, it must be installed. For a package `foo`, the name of the stub-only package (`foo-stubs`) is not a legal package name, so mypy will not find it, unless it is installed.

25.2 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"]
)
```

Note: If you use setuptools, you must pass the option `zip_safe=False` to `setup()`, or mypy will not be able to find the installed package.

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 interpreter 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/
```

(continues on next page)

(continued from previous page)

```
    __init__.pyi
    lib.pyi
```

the setup.py might look like:

```
from distutils.core import setup

setup(
    name="SuperPackageC",
    author="Me",
    version="0.1",
    package_data={"package_c-stubs": ["__init__.pyi", "lib.pyi"]},
    packages=["package_c-stubs"]
)
```

CHAPTER 26

Extending and integrating mypy

26.1 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])
```

26.2 Extending mypy using plugins

Python is a highly dynamic language and has extensive metaprogramming capabilities. Many popular libraries use these to create APIs that may be more flexible and/or natural for humans, but are hard to express using static types.

Extending the PEP 484 type system to accommodate all existing dynamic patterns is impractical and often just impossible.

Mypy supports a plugin system that lets you customize the way mypy type checks code. This can be useful if you want to extend mypy so it can type check code that uses a library that is difficult to express using just PEP 484 types.

The plugin system is focused on improving mypy's understanding of *semantics* of third party frameworks. There is currently no way to define new first class kinds of types.

Note: The plugin system is experimental and prone to change. If you want to write a mypy plugin, we recommend you start by contacting the mypy core developers on gitter. In particular, there are no guarantees about backwards compatibility. Backwards incompatible changes may be made without a deprecation period.

26.3 Configuring mypy to use plugins

Plugins are Python files that can be specified in a mypy *config file* using one of the two formats: relative or absolute path to the plugin to the plugin file, or a module name (if the plugin is installed using `pip install` in the same virtual environment where mypy is running). The two formats can be mixed, for example:

```
[mypy]
plugins = /one/plugin.py, other.plugin
```

Mypy will try to import the plugins and will look for an entry point function named `plugin`. If the plugin entry point function has a different name, it can be specified after colon:

```
[mypy]
plugins = custom_plugin:custom_entry_point
```

In following sections we describe basics of the plugin system with some examples. For more technical details please read docstrings in mypy/plugin.py in mypy source code. Also you can find good examples in the bundled plugins located in mypy/plugins.

26.4 High-level overview

Every entry point function should accept a single string argument that is a full mypy version and return a subclass of `mypy.plugins.Plugin`:

```
from mypy.plugin import Plugin

class CustomPlugin(Plugin):
    def get_type_analyze_hook(self, fullname: str):
        # see explanation below
        ...

def plugin(version: str):
    # ignore version argument if the plugin works with all mypy versions.
    return CustomPlugin
```

During different phases of analyzing the code (first in semantic analysis, and then in type checking) mypy calls plugin methods such as `get_type_analyze_hook()` on user plugins. This particular method for example can return a callback that mypy will use to analyze unbound types with given full name. See full plugin hook methods list *below*.

Mypy maintains a list of plugins it gets from the config file plus the default (built-in) plugin that is always enabled. Mypy calls a method once for each plugin in the list until one of the methods returns a non-`None` value. This callback will be then used to customize the corresponding aspect of analyzing/checking the current abstract syntax tree node.

The callback returned by the `get_xxx` method will be given a detailed current context and an API to create new nodes, new types, emit error messages etc., and the result will be used for further processing.

Plugin developers should ensure that their plugins work well in incremental and daemon modes. In particular, plugins should not hold global state due to caching of plugin hook results.

26.5 Current list of plugin hooks

get_type_analyze_hook() customizes behaviour of the type analyzer. For example, PEP 484 doesn't support defining variadic generic types:

```
from lib import Vector

a: Vector[int, int]
b: Vector[int, int, int]
```

When analyzing this code, mypy will call `get_type_analyze_hook("lib.Vector")`, so the plugin can return some valid type for each variable.

get_function_hook() is used to adjust the return type of a function call. This is a good choice if the return type of some function depends on *values* of some arguments that can't be expressed using literal types (for example a function may return an `int` for positive arguments and a `float` for negative arguments). This hook will be also called for instantiation of classes. For example:

```
from contextlib import contextmanager
from typing import TypeVar, Callable

T = TypeVar('T')

@contextmanager  # built-in plugin can infer a precise type here
def stopwatch(timer: Callable[[], T]) -> Iterator[T]:
    ...
    yield timer()
```

get_method_hook() is the same as `get_function_hook()` but for methods instead of module level functions.

get_method_signature_hook() is used to adjust the signature of a method. This includes special Python methods except `__init__()` and `__new__()`. For example in this code:

```
from ctypes import Array, c_int

x: Array[c_int]
x[0] = 42
```

mypy will call `get_method_signature_hook("ctypes.Array.__setitem__")` so that the plugin can mimic the `ctypes` auto-convert behavior.

get_attribute_hook overrides instance member field lookups and property access (not assignments, and not method calls). This hook is only called for fields which already exist on the class. *Exception:* if `__getattr__` or `__getattribute__` is a method on the class, the hook is called for all fields which do not refer to methods.

get_class_decorator_hook() can be used to update class definition for given class decorators. For example, you can add some attributes to the class to match runtime behaviour:

```
from lib import customize

@customize
class UserDefined:
    pass

var = UserDefined
var.customized  # mypy can understand this using a plugin
```

get_metaclass_hook() is similar to above, but for metaclasses.

get_base_class_hook() is similar to above, but for base classes.

get_dynamic_class_hook() can be used to allow dynamic class definitions in mypy. This plugin hook is called for every assignment to a simple name where right hand side is a function call:

```
from lib import dynamic_class

X = dynamic_class('X', [])
```

For such definition, mypy will call `get_dynamic_class_hook("lib.dynamic_class")`. The plugin should create the corresponding `mypy.nodes.TypeInfo` object, and place it into a relevant symbol table. (Instances of this class represent classes in mypy and hold essential information such as qualified name, method resolution order, etc.)

get_customize_class_mro_hook() can be used to modify class MRO (for example insert some entries there) before the class body is analyzed.

CHAPTER 27

Automatic stub generation (stubgen)

A stub file (see PEP 484) contains only type hints for the public interface of a module, with empty function bodies. Mypy can use a stub file instead of the real implementation to provide type information for the module. They are useful for third-party modules whose authors have not yet added type hints (and when no stubs are available in typeshed) and C extension modules (which mypy can't directly process).

Mypy includes the `stubgen` tool that can automatically generate stub files (`.pyi` files) for Python modules and C extension modules. For example, consider this source file:

```
from other_module import dynamic

BORDER_WIDTH = 15

class Window:
    parent = dynamic()
    def __init__(self, width, hight):
        self.width = width
        self.hight = hight

def create_empty() -> Window:
    return Window(0, 0)
```

Stubgen can generate this stub file based on the above file:

```
from typing import Any

BORDER_WIDTH: int = ...

class Window:
    parent: Any = ...
    width: Any = ...
    height: Any: ...
    def __init__(self, width, height) -> None: ...

def create_empty() -> Window: ...
```

Stubgen generates *draft* stubs. The auto-generated stub files often require some some manual updates, and most types will default to `Any`. The stubs will be much more useful if you add more precise type annotations, at least for the most commonly used functionality.

The rest of this section documents the command line interface of stubgen. Run `stubgen --help` for a quick summary of options.

Note: The command-line flags may change between releases.

27.1 Specifying what to stub

You can give stubgen paths of the source files for which you want to generate stubs:

```
$ stubgen foo.py bar.py
```

This generates stubs `out/foo.pyi` and `out/bar.pyi`. The default output directory `out` can be overridden with `-o DIR`.

You can also pass directories, and stubgen will recursively search them for any `.py` files and generate stubs for all of them:

```
$ stubgen my_pkg_dir
```

Alternatively, you can give module or package names using the `-m` or `-p` options:

```
$ stubgen -m foo -m bar -p my_pkg_dir
```

Details of the options:

`-m MODULE, --module MODULE` Generate a stub file for the given module. This flag may be repeated multiple times.

Stubgen *will not* recursively generate stubs for any submodules of the provided module.

`-p PACKAGE, --package PACKAGE` Generate stubs for the given package. This flag maybe repeated multiple times.

Stubgen *will* recursively generate stubs for all submodules of the provided package. This flag is identical to `--module` apart from this behavior.

Note: You can't mix paths and `-m`/`-p` options in the same stubgen invocation.

27.2 Specifying how to generate stubs

By default stubgen will try to import the target modules and packages. This allows stubgen to use runtime introspection to generate stubs for C extension modules and to improve the quality of the generated stubs. By default, stubgen will also use mypy to perform light-weight semantic analysis of any Python modules. Use the following flags to alter the default behavior:

`--no-import` Don't try to import modules. Instead use mypy's normal search mechanism to find sources. This does not support C extension modules. This flag also disables runtime introspection functionality, which mypy

uses to find the value of `__all__`. As result the set of exported imported names in stubs may be incomplete. This flag is generally only useful when importing a module generates an error for some reason.

`--parse-only` Don't perform semantic analysis of source files. This may generate worse stubs – in particular, some module, class, and function aliases may be represented as variables with the `Any` type. This is generally only useful if semantic analysis causes a critical mypy error.

`--doc-dir PATH` Try to infer better signatures by parsing .rst documentation in `PATH`. This may result in better stubs, but currently it only works for C extension modules.

27.3 Additional flags

`--py2` Run stubgen in Python 2 mode (the default is Python 3 mode).

`--ignore-errors` If an exception was raised during stub generation, continue to process any remaining modules instead of immediately failing with an error.

`--include-private` Include definitions that are considered private in stubs (with names such as `_foo` with single leading underscore and no trailing underscores).

`--search-path PATH` Specify module search directories, separated by colons (only used if `--no-import` is given).

`--python-executable PATH` Use Python interpreter at `PATH` for importing modules and runtime introspection. This has no effect with `--no-import`, and this only works in Python 2 mode. In Python 3 mode the Python interpreter used to run stubgen will always be used.

`-o PATH`, `--output PATH` Change the output directory. By default the stubs are written in the `./out` directory. The output directory will be created if it doesn't exist. Existing stubs in the output directory will be overwritten without warning.

CHAPTER 28

Common issues and solutions

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 (except for type annotations), but sometimes you need to do things slightly differently.

28.1 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`.

28.2 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).

28.3 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.

Another option is to explicitly annotate values with type `Any` – mypy will let you perform arbitrary operations on `Any` values. Sometimes there is no more precise type you can use for a particular value, especially if you use dynamic Python features such as `__getattr__`:

```
class Wrapper:
    ...
    def __getattr__(self, a: str) -> Any:
        return getattr(self._wrapped, a)
```

Finally, you can create a stub file (`.pyi`) for a file that generates spurious errors. Mypy will only look at the stub file and ignore the implementation, since stub files take precedence over `.py` files.

28.4 Unexpected errors about 'None' and/or 'Optional' types

Starting from mypy 0.600, mypy uses *strict optional checking* by default, and the `None` value 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*). You can also fall back to this behavior if strict optional checking would require a large number of `assert foo is not None` checks to be inserted, and you want to minimize the number of code changes required to get a clean mypy run.

28.5 Mypy runs are slow

If your mypy runs feel slow, you should probably use the *mypy daemon*, which can speed up incremental mypy runtimes by a factor of 10 or more. *Remote caching* can make cold mypy runs several times faster.

28.6 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: 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!

28.7 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!
```

28.8 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
```

28.9 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
```

28.10 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.

28.11 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
```

As a special case, you can also use one of these checks in a top-level (unindented) `assert`; this makes mypy skip the rest of the file. Example:

```
import sys

assert sys.platform != 'win32'

# The rest of this file doesn't apply to Windows.
```

Some other expressions exhibit similar behavior; in particular, `typing.TYPE_CHECKING`, variables named `MYPY`, and any variable whose name is passed to `--always-true` or `--always-false`. (However, `True` and `False` are not treated specially!)

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.

28.12 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 variables 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.

28.13 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]
```

28.14 Using classes that are generic in stubs but not at runtime

Some classes are declared as generic in stubs, but not at runtime. Examples in the standard library include `os.PathLike` and `queue.Queue`. Subscripting such a class will result in a runtime error:

```
from queue import Queue

class Tasks(Queue[str]):  # TypeError: 'type' object is not subscriptable
    ...

results: Queue[int] = Queue()  # TypeError: 'type' object is not subscriptable
```

To avoid these errors while still having precise types you can either use string literal types or `typing.TYPE_CHECKING`:

```
from queue import Queue
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    BaseQueue = Queue[str]  # this is only processed by mypy
else:
    BaseQueue = Queue  # this is not seen by mypy but will be executed at runtime.

class Tasks(BaseQueue):  # OK
    ...

results: 'Queue[int]' = Queue()  # OK
```

28.15 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
```

28.16 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
```

28.17 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_):
        ...
```

28.18 I need a mypy bug fix that hasn't been released yet

You can install the latest development version of 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 29

Supported Python features

A list of unsupported Python features is maintained in the mypy wiki:

- Unsupported Python features

29.1 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 30

New features in Python 3.6

Mypy has supported all language features new in Python 3.6 starting with mypy 0.510. This section introduces Python 3.6 features that interact with type checking.

30.1 Syntax for variable annotations (PEP 526)

Python 3.6 introduced a new syntax for variable annotations (in global, class and local scopes). There are two variants of the syntax, with or without an initializer expression:

```
from typing import Optional
foo: Optional[int]  # No initializer
bar: List[str] = []  # Initializer
```

You can also 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
```

30.2 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. It also introduced a syntax for asynchronous comprehensions. This example uses the `AsyncIterator` type to define an async generator:

```
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")
```

30.3 New named tuple syntax

Python 3.6 supports an alternative, class-based syntax for named tuples. See *Named tuples* for the details.

CHAPTER 31

Additional features

This section discusses various features that did not fit in naturally in one of the previous sections.

31.1 Dataclasses

In Python 3.7, a new `dataclasses` module has been added to the standard library. This module allows defining and customizing simple boilerplate-free classes. They can be defined using the `@dataclasses.dataclass` decorator:

```
from dataclasses import dataclass, field

@dataclass
class Application:
    name: str
    plugins: List[str] = field(default_factory=list)

test = Application("Testing...")  # OK
bad = Application("Testing...", "with plugin")  # Error: List[str] expected
```

Mypy will detect special methods (such as `__lt__`) depending on the flags used to define dataclasses. For example:

```
from dataclasses import dataclass

@dataclass(order=True)
class OrderedPoint:
    x: int
    y: int

@dataclass(order=False)
class UnorderedPoint:
    x: int
    y: int
```

(continues on next page)

(continued from previous page)

```
OrderedPoint(1, 2) < OrderedPoint(3, 4)  # OK
UnorderedPoint(1, 2) < UnorderedPoint(3, 4)  # Error: Unsupported operand types
```

Dataclasses can be generic and can be used in any other way a normal class can be used:

```
from dataclasses import dataclass
from typing import Generic, TypeVar

T = TypeVar('T')

@dataclass
class BoxedData(Generic[T]):
    data: T
    label: str

def unbox(bd: BoxedData[T]) -> T:
    ...

val = unbox(BoxedData(42, "<important>"))  # OK, inferred type is int
```

For more information see official docs and PEP 557.

31.1.1 Caveats/Known Issues

Some functions in the `dataclasses` module, such as `replace()` and `asdict()`, have imprecise (too permissive) types. This will be fixed in future releases.

Mypy does not yet recognize aliases of `dataclasses.dataclass`, and will probably never recognize dynamically computed decorators. The following examples do **not** work:

```
from dataclasses import dataclass

dataclass_alias = dataclass
def dataclass_wrapper(cls):
  return dataclass(cls)

@dataclass_alias
class AliasDecorated:
  """
  Mypy doesn't recognize this as a dataclass because it is decorated by an
  alias of `dataclass` rather than by `dataclass` itself.
  """
  attribute: int

@dataclass_wrapper
class DynamicallyDecorated:
  """
  Mypy doesn't recognize this as a dataclass because it is decorated by a
  function returning `dataclass` rather than by `dataclass` itself.
  """
  attribute: int

AliasDecorated(attribute=1) # error: Unexpected keyword argument
DynamicallyDecorated(attribute=1) # error: Unexpected keyword argument
```

31.2 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:
    one: int = attr.ib(8)
    two: Dict[str, str] = attr.Factory(dict)
    bad: str = attr.ib(16)    # Error: can't assign int to str
```

31.2.1 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.

31.3 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.

31.3.1 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.

31.3.2 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.

31.3.3 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!

31.3.4 Caching with mypy daemon

You can also use remote caching with the *mypy daemon*. The remote cache will significantly speed up 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 `dmypy 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.

31.3.5 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).

31.4 Extended Callable types

As an experimental mypy extension, you can specify `Callable` types that support keyword arguments, optional arguments, and more. When 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.

CHAPTER 32

Frequently Asked Questions

32.1 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 or static typing for Python.

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.

32.2 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 helps you find errors quickly 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. The earlier you start, the easier it will be to adopt static typing.

32.3 Can I use mypy to type check my existing Python code?

Mypy supports most Python features and idioms, and many large Python projects are using mypy successfully. 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 codebase that are less dynamic.

32.4 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.

32.5 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`.

32.6 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.

32.7 Can I use duck typing with mypy?

Mypy provides support for both nominal subtyping and structural subtyping. Structural subtyping can be thought of as "static duck typing". 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.

32.8 I like Python and I have no need for static typing

The aim of mypy is not to convince everybody to write statically typed Python – static typing is entirely optional, now and in the future. 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.

32.9 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 and solutions* mentions some modifications to Python code that may be required to make code type check without errors. Also, your code must make attributes explicit.

Mypy supports modular, efficient type checking, and this seems to rule out type checking some language features, such as arbitrary monkey patching of methods.

32.10 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.

32.11 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 33

Indices and tables

- genindex
- search