# Contents

1. **Getting started** .................................................. 3
2. **Using mypy with an existing codebase** ....................... 10
3. **Type hints cheat sheet (Python 3)** ........................... 13
4. **Built-in types** .................................................. 19
5. **Type inference and type annotations** ......................... 20
6. **Kinds of types** .................................................. 24
7. **Class basics** .................................................. 35
8. **Annotation issues at runtime** ................................ 40
9. **Protocols and structural subtyping** ........................ 44
10. **Dynamically typed code** ...................................... 52
11. **Type narrowing** ................................................ 53
12. **Duck type compatibility** ..................................... 59
13. **Stub files** .................................................. 60
14. **Generics** .................................................. 61
15. **More types** .................................................. 75
16. **Literal types and Enums** .................................... 93
17. **Final names, methods and classes** .......................... 101
18. **Metaclasses** .................................................. 105
19. **Running mypy and managing imports** ....................... 106
20. **The mypy command line** .................................... 113
21. **The mypy configuration file** ................................ 126
22. **Inline configuration** .......................................... 142
23. **Mypy daemon (mypy server)** ................................ 142
24. **Using installed packages** ..................................... 147
25. **Extending and integrating mypy** ............................ 150
26. **Automatic stub generation (stubgen)** ....................... 154
27. **Automatic stub testing (stubtest)** .......................... 156
28. **Common issues and solutions** ................................ 158
29. **Supported Python features** .................................. 170
30. **Error codes** ................................................ 170
31. **Error codes enabled by default** ................................ 172
32. **Error codes for optional checks** ............................. 183
33. **Additional features** .......................................... 189
34. **Frequently Asked Questions** ................................ 195

## Indices and tables

2. **Indices and tables** ............................................. 199

**Index** .................................................................. 201
Mypy is a static type checker for Python 3. 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 annotation syntax (using PEP 484 and PEP 526 notation), 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.
1.1 Getting started

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

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

1.1.1 Installing and running mypy

Mypy requires Python 3.6 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.

1.1.2 Function signatures and dynamic vs static typing

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

```python
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:

```python
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.

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.

### 1.1.3 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:

```python
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:

```python
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 = f'Hello, {name}'
    if excited:
        message += '!!!'
    return message

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

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

1.1.4 Additional types, and 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”?

For example, to indicate that some function can accept a list of strings, use the list[str] type (Python 3.9 and later):

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 Python 3.8 and earlier, you can instead import the List type from the typing module:

from typing import List  # Python 3.8 and earlier

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

...
from collections.abc import Iterable  # or "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 f'user-{100_000 + 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 cheatsheet 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.

Note: When adding types, the convention is to import types using the form from typing import Union (as opposed to doing just import typing or import typing as t or from typing import *).

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

Note: In some examples we use capitalized variants of types, such as List, and sometimes we use plain list. They are equivalent, but the prior variant is needed if you are using Python 3.8 or earlier.
1.1.5 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:

```python
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:

```python
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 PEP 526) or using a comment-based syntax like so:

```python
# If you're using Python 3.9+
my_global_dict: dict[int, float] = {}

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

1.1.6 Types and classes

So far, we’ve only seen examples of pre-existing types like the int or float builtins, or generic types from collections.abc and typing, such as Iterable. However, these aren’t the only types you can use: in fact, you can use any Python class as a type!

For example, suppose you’ve defined a custom class representing a bank account:

```python
class BankAccount:
    # Note: It is ok to omit type hints for the "self" parameter.
    # Mypy will infer the correct type.
    def __init__(self, account_name: str, initial_balance: int = 0) -> None:
        # Note: Mypy will infer the correct types of your fields
        # based on the types of the parameters.
        self.account_name = account_name
        self.balance = initial_balance

    def deposit(self, amount: int) -> None:
```

(continues on next page)
You can declare that a function will accept any instance of your class by simply annotating the parameters with `BankAccount`:

```python
def transfer(src: BankAccount, dst: BankAccount, amount: int) -> None:
    src.withdraw(amount)
    dst.deposit(amount)
```

```python
account_1 = BankAccount('Alice', 400)
account_2 = BankAccount('Bob', 200)
transfer(account_1, account_2, 50)
```

In fact, the `transfer` function we wrote above can accept more than just instances of `BankAccount`: it can also accept any instance of a subclass of `BankAccount`. For example, suppose you write a new class that looks like this:

```python
class AuditedBankAccount(BankAccount):
    def __init__(self, account_name: str, initial_balance: int = 0) -> None:
        super().__init__(account_name, initial_balance)
        self.audit_log: list[str] = []

    def deposit(self, amount: int) -> None:
        self.audit_log.append(f'Deposited {amount}"
        self.balance += amount

    def withdraw(self, amount: int) -> None:
        self.audit_log.append(f'Withdraw {amount}"
        self.balance -= amount
```

Since `AuditedBankAccount` is a subclass of `BankAccount`, we can directly pass in instances of it into our `transfer` function:

```python
audited = AuditedBankAccount('Charlie', 300)
transfer(account_1, audited, 100)  # Type checks!
```

This behavior is actually a fundamental aspect of the PEP 484 type system: when we annotate some variable with a type `T`, we are actually telling mypy that variable can be assigned an instance of `T`, or an instance of a subclass of `T`. The same rule applies to type hints on parameters or fields.

See `Class basics` to learn more about how to work with code involving classes.
1.1.7 Stubs files and typeshed

Mypy also understands how to work with classes found in the standard library. For example, here is a function which uses the `Path` object from the `pathlib` standard library module:

```python
from pathlib import Path

def load_template(template_path: Path, name: str) -> str:
    # Mypy understands that 'file_path.read_text()' returns a str...
    template = template_path.read_text()

    # ...so understands this line type checks.
    return template.replace('USERNAME', name)
```

This behavior may surprise you if you're familiar with how Python internally works. The standard library does not use type hints anywhere, so how did mypy know that `Path.read_text()` returns a `str`, or that `str.replace(...)` accepts exactly two `str` arguments?

The answer is that mypy comes bundled with stub files from the the typeshed project, which contains stub files for the Python builtins, the standard library, and selected third-party packages.

A stub file is a file containing a skeleton of the public interface of that Python module, including classes, variables, functions – and most importantly, their types.

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 or inline annotations that mypy can automatically find, or you can install additional stubs using pip (see Missing imports and Using installed packages for the details). For example, you can install the stubs for the `requests` package like this:

```
$ python3 -m pip install types-requests
```

The stubs are usually packaged in a distribution named `types-<distribution>`. Note that the distribution name may be different from the name of the package that you import. For example, `types-PyYAML` contains stubs for the `yaml` package. Mypy can often suggest the name of the stub distribution:

```
prog.py:1: error: Library stubs not installed for "yaml" (or incompatible with Python 3. ˓→8)
prog.py:1: note: Hint: "python3 -m pip install types-PyYAML"
...
```

You can also create stubs easily. We discuss strategies for handling errors about missing stubs in Missing imports.

1.1.8 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 (though 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.

### 1.1.9 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*.
- 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.

### 1.2 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.

#### 1.2.1 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:
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`:

```python
import frobnicate  # type: ignore
...
```

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.

### 1.2.2 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.
1.2.3 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==0.790 # Pinned version avoids surprises
scripts/mypy # Run the mypy runner script you set up
```

1.2.4 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.

1.2.5 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.

1.2.6 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.

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.

1.2.7 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.
1.2.8 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 using `disallow_untyped_defs`, or type check code without annotations as well with `check_untyped_defs`. Refer to `The mypy configuration file` for the details.

1.3 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.

1.3.1 Variables

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

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

# 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
```

1.3.2 Built-in types

```python
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 type of the collection item is in brackets
# (Python 3.9+)
x: list[int] = [1]
```

(continues on next page)
x: set[int] = {6, 7}

# In Python 3.8 and earlier, the name of the collection type is
# capitalized, and the type is imported from the 'typing' module
x: List[int] = [1]
x: Set[int] = {6, 7}

# For mappings, we need the types of both keys and values
x: dict[str, float] = {"field": 2.0}  # Python 3.9+
x: Dict[str, float] = {"field": 2.0}

# For tuples of fixed size, we specify the types of all the elements
x: tuple[int, str, float] = (3, "yes", 7.5)  # Python 3.9+
x: Tuple[int, str, float] = (3, "yes", 7.5)

# For tuples of variable size, we use one type and ellipsis
x: tuple[int, ...] = (1, 2, 3)  # Python 3.9+
x: Tuple[int, ...] = (1, 2, 3)

# 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())

## 1.3.3 Functions

Python 3 supports an annotation syntax for function declarations.

```python
from typing import Callable, Iterator, Union, Optional

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

# A generator function that yields ints is secretly just a function that
# returns an iterator of ints, so that's how we annotate it
```

(continues on next page)
def g(n: int) -> Iterator[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

1.3.4 When you’re puzzled or when things are complicated

from typing import Union, Any, 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,
# 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.

```python
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.

```python
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

## 1.3.5 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.

```python
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(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_mapping: Mapping[int, str]) -> list[int]:
    my_mapping[5] = 'maybe'  # if we try this, mypy will throw an error...
    return list(my_mapping.keys())
```

(continues on next page)
You can even make your own duck types using *Protocols and structural subtyping*.

### 1.3.6 Classes

```python
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

```python
x: MyClass = MyClass()
```

# You can use the ClassVar annotation to declare a class variable

```python
class Car:
    seats: ClassVar[int] = 4
    passengers: ClassVar[list[str]]
```

# You can also declare the type of an attribute in "__init__"

```python
class Box:
    def __init__(self) -> None:
        self.items: list[str] = []
```
1.3.7 Coroutines and asyncio

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

```python
import asyncio

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

1.3.8 Miscellaneous

```python
import sys
import re
from typing import Match, 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
    ...
```
1.3.9 Decorators

Decorator functions can be expressed via generics. See *Declaring decorators* for more details.

```python
from typing import Any, Callable, TypeVar

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

def bare_decorator(func: F) -> F:
    ...

def decorator_args(url: str) -> Callable[[F], F]:
    ...
```

1.4 Built-in types

This chapter introduces some commonly used built-in types. We will cover many other kinds of types later.

1.4.1 Simple types

Here are examples of some common built-in types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>float</td>
<td>floating point number</td>
</tr>
<tr>
<td>bool</td>
<td>boolean value (subclass of int)</td>
</tr>
<tr>
<td>str</td>
<td>string (unicode in Python 3)</td>
</tr>
<tr>
<td>bytes</td>
<td>8-bit string</td>
</tr>
<tr>
<td>object</td>
<td>an arbitrary object (object is the common base class)</td>
</tr>
</tbody>
</table>

All built-in classes can be used as types.

1.4.2 Any type

If you can’t find a good type for some value, you can always fall back to `Any`:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>dynamically typed value with an arbitrary type</td>
</tr>
</tbody>
</table>

The type `Any` is defined in the `typing` module. See *Dynamically typed code* for more details.
1.4.3 Generic types

In Python 3.9 and later, built-in collection type objects support indexing:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list[str]</td>
<td>list of str objects</td>
</tr>
<tr>
<td>tuple[int, int]</td>
<td>tuple of two int objects (tuple[()] is the empty tuple)</td>
</tr>
<tr>
<td>tuple[int, ...]</td>
<td>tuple of an arbitrary number of int objects</td>
</tr>
<tr>
<td>dict[str, int]</td>
<td>dictionary from str keys to int values</td>
</tr>
<tr>
<td>Iterable[int]</td>
<td>iterable object containing ints</td>
</tr>
<tr>
<td>Sequence[bool]</td>
<td>sequence of booleans (read-only)</td>
</tr>
<tr>
<td>Mapping[str, int]</td>
<td>mapping from str keys to int values (read-only)</td>
</tr>
<tr>
<td>type[C]</td>
<td>type object of C (C is a class/type variable/union of types)</td>
</tr>
</tbody>
</table>

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.

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. You can import them from collections.abc instead of importing from typing in Python 3.9.

See Using generic builtins for more details, including how you can use these in annotations also in Python 3.7 and 3.8. These legacy types defined in typing are needed if you need to support Python 3.8 and earlier:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List[str]</td>
<td>list of str objects</td>
</tr>
<tr>
<td>Tuple[int, int]</td>
<td>tuple of two int objects (Tuple[()] is the empty tuple)</td>
</tr>
<tr>
<td>Tuple[int, ...]</td>
<td>tuple of an arbitrary number of int objects</td>
</tr>
<tr>
<td>Dict[str, int]</td>
<td>dictionary from str keys to int values</td>
</tr>
<tr>
<td>Iterable[int]</td>
<td>iterable object containing ints</td>
</tr>
<tr>
<td>Sequence[bool]</td>
<td>sequence of booleans (read-only)</td>
</tr>
<tr>
<td>Mapping[str, int]</td>
<td>mapping from str keys to int values (read-only)</td>
</tr>
<tr>
<td>type[C]</td>
<td>type object of C (C is a class/type variable/union of types)</td>
</tr>
</tbody>
</table>

List is an alias for the built-in type list that supports indexing (and similarly for dict/Dict and tuple/Tuple).

Note that even though Iterable, Sequence and Mapping look similar to abstract base classes defined in collections.abc (formerly collections), they are not identical, since the latter don’t support indexing prior to Python 3.9.

1.5 Type inference and type annotations

1.5.1 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:

```python
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.

### 1.5.2 Explicit types for variables

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

```python
go from typing import Union
g 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:

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

**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>)`.

### 1.5.3 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:

```python
go l = [] # Error: Need type annotation for "l"
```

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

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

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

```python
go s: set[int] = set()
```

**Note:** Using type arguments (e.g. `list[int]`) on builtin collections like `list`, `dict`, `tuple`, and `set` only works in Python 3.9 and later. For Python 3.8 and earlier, you must use `List` (e.g. `List[int]`), `Dict`, and so on.
1.5.4 Compatibility of container types

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

```python
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:

```python
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.

1.5.5 Context in type inference

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

```python
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:

```python
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:

```python
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:
1.5.6 Starred expressions

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

```python
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:

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

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

1.5.7 Silencing type errors

You might want to disable type checking on specific lines, or within specific files in your codebase. To do that, you can use a `# type: ignore` comment.

For example, say that the web framework that you use now takes an integer argument to `run()`, which starts it on localhost on that port. Like so:

```python
# Starting app on http://localhost:8000
app.run(8000)
```

However, the type stubs that the package uses is not up-to-date, and it still expects only `str` types for `run()`. This would give you the following error:

```
error: Argument 1 to "run" of "A" has incompatible type "int"; expected "str"
```

If you cannot directly fix the type stubs yourself, you can temporarily disable type checking on that line, by adding a `# type: ignore`:

```python
# Starting app on http://localhost:8000
app.run(8000)  # type: ignore
```

This will suppress any mypy errors that would have raised on that specific line.

You should probably add some more information on the `# type: ignore` comment, to explain why the ignore was added in the first place. This could be a link to an issue on the repository responsible for the type stubs, or it could be a short explanation of the bug. To do that, use this format:

```python
# Starting app on http://localhost:8000
app.run(8000)  # type: ignore  # `run()` now accepts an `int`, as a port
```

Mypy displays an error code for each error if you use `--show-error-codes`.
error: "str" has no attribute "trim" [attr-defined]

It is possible to add a specific error-code in your ignore comment (e.g. # type: ignore[attr-defined]) to clarify what’s being silenced. You can find more information about error codes here.

Similarly, you can also ignore all mypy checks in a file, by adding a # type: ignore at the top of the file:

```python
# type: ignore
# This is a test file, skipping type checking in it.
import unittest
...
```

Finally, adding a @typing.no_type_check decorator to a class, method or function has the effect of ignoring that class, method or function:

```python
@typing.no_type_check
def foo() -> str:
    return 12345 # No error!
```

### 1.6 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.

#### 1.6.1 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:

```python
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)
```
1.6.2 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:

```python
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:

```python
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:

```python
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:

```python
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.
1.6.3 Tuple types

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

```python
# Use `typing.Tuple` in Python 3.8 and earlier

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:

```python
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).

1.6.4 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:

```python
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:
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.

### 1.6.5 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.
1.6.6 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]`):

```python
from typing import Optional

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

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

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

```python
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:

```python
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:

```python
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:

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

    def initialize(self, path: str) -> None:
        self.path = path
```
(continues on next page)
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 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

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

class Container:
    items: list[str]  # No initial value

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. (NoneType is available as types.NoneType on Python 3.10+, but is not exposed at all on earlier versions of Python.)

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.

X | Y syntax for Unions

PEP 604 introduced an alternative way for spelling union types. In Python 3.10 and later, you can write Union[int, str] as int | str. It is possible to use this syntax in versions of Python where it isn’t supported by the runtime with some limitations (see Annotation issues at runtime).

```python
t1: int | str  # equivalent to Union[int, str]
t2: int | None  # equivalent to Optional[int]
```

1.6.7 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:

```python
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:

```python
def greeting(name: str) -> str:
    if name:
        return f'Hello, {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:

```python
from typing import Optional
```

(continues on next page)
def greeting(name: Optional[str]) -> str:
    if name:
        return f'Hello, {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.

### 1.6.8 Type aliases

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

```python
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:

```python
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.

Since Mypy 0.930 you can also use *explicit type aliases*, which were introduced in PEP 613.

There can be confusion about exactly when an assignment defines an implicit type alias – for example, when the alias contains forward references, invalid types, or violates some other restrictions on type alias declarations. Because the distinction between an unannotated variable and a type alias is implicit, ambiguous or incorrect type alias declarations default to defining a normal variable instead of a type alias.

Explicit type aliases are unambiguous and can also improve readability by making the intent clear:

```python
from typing import TypeAlias  # "from typing_extensions" in Python 3.9 and earlier

AliasType: TypeAlias = Union[list[dict[tuple[int, str], set[int]]], tuple[str, list[str]]]
```
1.6.9 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:

```python
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 `NamedTuple` to also define item types:

```python
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:

```python
from typing import NamedTuple
class Point(NamedTuple):
    x: int
    y: int
p = Point(x=1, y='x')  # Argument has incompatible type "str"; expected "int"
```

Note: You can use the raw `NamedTuple` “pseudo-class” in type annotations if any `NamedTuple` object is valid. For example, it can be useful for deserialization:

```python
def deserialize_named_tuple(arg: NamedTuple) -> Dict[str, Any]:
    return arg._asdict()
Point = namedtuple('Point', ['x', 'y'])
Person = NamedTuple('Person', [('name', str), ('age', int)])
deserialize_named_tuple(Point(x=1, y=2))  # ok
deserialize_named_tuple(Person(name='Nikita', age=18))  # ok
# Error: Argument 1 to "deserialize_named_tuple" has incompatible type
# "Tuple[int, int]"; expected "NamedTuple"
deserialize_named_tuple((1, 2))
```

Note that this behavior is highly experimental, non-standard, and may not be supported by other type checkers and IDEs.
1.6.10 The type of class objects

(Freely after PEP 484: The type of class objects.)

Sometimes you want to talk about class objects that inherit from a given class. This can be spelled as \texttt{type[C]} (or, on Python 3.8 and lower, \texttt{typing.Type[C]} where \texttt{C} is a class. In other words, when \texttt{C} is the name of a class, using \texttt{C} to annotate an argument declares that the argument is an instance of \texttt{C} (or of a subclass of \texttt{C}), but using \texttt{type[C]} as an argument annotation declares that the argument is a class object deriving from \texttt{C} (or \texttt{C} itself).

For example, assume the following classes:

```python
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 \texttt{ProUser} doesn’t inherit from \texttt{BasicUser}.

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

```python
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 the ability to parameterize \texttt{type}, the best we could do would be:

```python
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 \texttt{buyer} variable has type \texttt{ProUser}:

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

However, using the \texttt{type[C]} syntax and a type variable with an upper bound (see \texttt{Type variables with upper bounds}) we can do better:

```python
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 \texttt{new_user()} with a specific subclass of \texttt{User}:

```python
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[] and typing.Type[], see PEP 484: The type of class objects.

1.6.11 Generators

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

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

A good rule of thumb is to annotate functions with the most specific return type possible. However, you should also take care to avoid leaking implementation details into a function’s public API. In keeping with these two principles, prefer Iterator[YieldType] over Iterable[YieldType] as the return-type annotation for a generator function, as it lets mypy know that users are able to call next() on the object returned by the function. Nonetheless, bear in mind that Iterable may sometimes be the better option, if you consider it an implementation detail that next() can be called on the object returned by your function.

If you want your generator to accept values via the send() method or return a value, on the other hand, you should use the Generator[YieldType, SendType, ReturnType] generic type instead of either Iterator or Iterable. For example:

```python
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:

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

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

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

1.7.1 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:

```python
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:

```python
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.)

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

```python
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`:

```python
class A:
    def __init__(self) -> None:
        self.y = 1  # Define 'y'
        a = self
        a.x = 1    # Error: 'x' not defined
```
1.7.2 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:

```python
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:

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

1.7.3 Class attribute annotations

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

```python
from typing import ClassVar

class A:
    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
```

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:

```python
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.

An explicit ClassVar may be particularly handy to distinguish between class and instance variables with callable types. For example:

```python
from typing import Callable, ClassVar

class A:
    foo: Callable[[int], None]
    bar: ClassVar[Callable[[A, int], None]]
    bad: Callable[[A], None]
    
A().foo(42)  # OK
A().bar(42)  # OK
A().bad()    # Error: Too few arguments
```

**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).

### 1.7.4 Overriding statically typed methods

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

```python
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):
    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:

```python
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
```

### 1.7.5 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:

```python
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

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

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

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

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):

```python
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:

```python
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:

```python
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.

### 1.7.6 Slots

When a class has explicitly defined `__slots__`, `mypy` will check that all attributes assigned to are members of `__slots__`:

```python
class Album:
    __slots__ = ('name', 'year')

    def __init__(self, name: str, year: int) -> None:
        self.name = name
        self.year = year
        # Error: Trying to assign name "released" that is not in "__slots__" of type "Album"
        self.released = True

my_album = Album('Songs about Python', 2021)
```

`Mypy` will only check attribute assignments against `__slots__` when the following conditions hold:

1. All base classes (except builtin ones) must have explicit `__slots__` defined (this mirrors Python semantics).
2. `__slots__` does not include `__dict__`. If `__slots__` includes `__dict__`, arbitrary attributes can be set, similar to when `__slots__` is not defined (this mirrors Python semantics).
3. All values in `__slots__` must be string literals.
1.8 Annotation issues at runtime

Idiomatic use of type annotations can sometimes run up against what a given version of Python considers legal code. This section describes these scenarios and explains how to get your code running again. Generally speaking, we have three tools at our disposal:

- For Python 3.7 through 3.9, use of `from __future__ import annotations` (PEP 563), made the default in Python 3.11 and later
- Use of string literal types or type comments
- Use of `typing.TYPE_CHECKING`

We provide a description of these before moving onto discussion of specific problems you may encounter.

1.8.1 String literal types

Type comments can’t cause runtime errors because comments are not evaluated by Python. In a similar way, using string literal types sidesteps the problem of annotations that would cause runtime errors.

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

```python
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 and stub files.

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.)

1.8.2 Future annotations import (PEP 563)

Many of the issues described here are caused by Python trying to evaluate annotations. From Python 3.11 on, Python will no longer attempt to evaluate function and variable annotations. This behaviour is made available in Python 3.7 and later through the use of `from __future__ import annotations`.

This can be thought of as automatic string literal-ification of all function and variable annotations. Note that function and variable annotations are still required to be valid Python syntax. For more details, see PEP 563.

Note: Even with the `__future__ import`, there are some scenarios that could still require string literals or result in errors, typically involving use of forward references or generics in:

- `type aliases;`
- `type narrowing;`
- `type definitions (see TypeVar, NewType(), NamedTuple);`
- `base classes.`

```python
# base class example
from __future__ import annotations
class A(tuple['B', 'C']): ... # String literal types needed here
class B: ...
class C: ...
```
Note: Some libraries may have use cases for dynamic evaluation of annotations, for instance, through use of typing. get_type_hints or eval. If your annotation would raise an error when evaluated (say by using PEP 604 syntax with Python 3.9), you may need to be careful when using such libraries.

1.8.3 typing.TYPE_CHECKING

The typing module defines a TYPE_CHECKING constant that is False at runtime but treated as True while type checking.

Since code inside if TYPE_CHECKING: is not executed at runtime, it provides a convenient way to tell mypy something without the code being evaluated at runtime. This is most useful for resolving import cycles.

1.8.4 Class name forward references

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

```python
def f(x: A) -> None: ...  # NameError: name "A" is not defined
class A: ...
```

Starting from Python 3.7, you can add from __future__ import annotations to resolve this, as discussed earlier:

```python
from __future__ import annotations
def f(x: A) -> None: ...  # OK
class A: ...
```

For Python 3.6 and below, you can enter the type as a string literal or type comment:

```python
def f(x: 'A') -> None: ...  # OK
# Also OK
def g(x):  # type: (A) -> None  
    ...
class A: ...
```

Of course, instead of using future annotations import or string literal types, you could move the function definition after the class definition. This is not always desirable or even possible, though.
1.8.5 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. This can lead to errors at runtime like:

```
ImportError: cannot import name 'b' from partially initialized module 'A' (most likely...
˓due to a circular import)
```

If those cycles do become a problem when running your program, there’s a trick: if the import is only needed for type annotations and you’re using a) the `future annotations import`, or b) string literals or type comments for the relevant annotations, you can write the imports inside `if TYPE_CHECKING:` so that they are not executed at runtime. Example:

File `foo.py`:

```python
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    import bar

def listify(arg: 'bar.BarClass') -> 'list[bar.BarClass]':
    return [arg]
```

File `bar.py`:

```python
from foo import listify

class BarClass:
    def listifyme(self) -> 'list[BarClass]':
        return listify(self)
```

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

Some classes are declared as `generic` in stubs, but not at runtime.

In Python 3.8 and earlier, there are several examples within the standard library, for instance, `os.PathLike` and `queue.Queue`. Subscripting such a class will result in a runtime error:

```python
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 errors from use of these generics in annotations, just use the `future annotations import` (or string literals or type comments for Python 3.6 and below).

To avoid errors when inheriting from these classes, things are a little more complicated and you need to use `typing.TYPE_CHECKING`:

```python
from typing import TYPE_CHECKING
from queue import Queue
```

(continues on next page)
If your subclass is also generic, you can use the following:

```python
from typing import TYPE_CHECKING, TypeVar, Generic
from queue import Queue

_T = TypeVar("_T")
if TYPE_CHECKING:
    class _MyQueueBase(Queue[_T]): pass
else:
    class _MyQueueBase(Generic[_T], Queue): pass

class MyQueue(_MyQueueBase[_T]): pass

task_queue: MyQueue[str]
reveal_type(task_queue.get())  # Reveals str
```

In Python 3.9, we can just inherit directly from `Queue[str]` or `Queue[T]` since its `queue.Queue` implements `__class_getitem__()`, so the class object can be subscripted at runtime without issue.

### 1.8.7 Using types defined in stubs but not at runtime

Sometimes stubs that you’re using may define types you wish to re-use that do not exist at runtime. Importing these types naively will cause your code to fail at runtime with `ImportError` or `ModuleNotFoundError`. Similar to previous sections, these can be dealt with by using `typing.TYPE_CHECKING`:

```python
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from _typeshed import SupportsRichComparison
```

### 1.8.8 Using generic builtins

Starting with Python 3.9 (PEP 585), the type objects of many collections in the standard library support subscription at runtime. This means that you no longer have to import the equivalents from `typing`; you can simply use the built-in collections or those from `collections.abc`:

```python
from collections.abc import Sequence
x: list[str]
y: dict[int, str]
z: Sequence[str] = x
```
There is limited support for using this syntax in Python 3.7 and later as well. If you use `from __future__ import annotations`, mypy will understand this syntax in annotations. However, since this will not be supported by the Python interpreter at runtime, make sure you're aware of the caveats mentioned in the notes at `future annotations import`.

### 1.8.9 Using X | Y syntax for Unions

Starting with Python 3.10 ([PEP 604](https://www.python.org/dev/peps/pep-0604/)), you can spell union types as `x: int | str`, instead of `x: typing.Union[int, str].`

There is limited support for using this syntax in Python 3.7 and later as well. If you use `from __future__ import annotations`, mypy will understand this syntax in annotations, string literal types, type comments and stub files. However, since this will not be supported by the Python interpreter at runtime (if evaluated, `int | str` will raise `TypeError: unsupported operand type(s) for |: 'type' and 'type'`), make sure you're aware of the caveats mentioned in the notes at `future annotations import`.

### 1.8.10 Using new additions to the typing module

You may find yourself wanting to use features added to the `typing` module in earlier versions of Python than the addition, for example, using any of `Literal`, `Protocol`, `TypedDict` with Python 3.6.

The easiest way to do this is to install and use the `typing_extensions` package from PyPI for the relevant imports, for example:

```python
from typing_extensions import Literal
x: Literal["open", "close"]
```

If you don't want to rely on `typing_extensions` being installed on newer Pythons, you could alternatively use:

```python
import sys
if sys.version_info >= (3, 8):
    from typing import Literal
else:
    from typing_extensions import Literal

x: Literal["open", "close"]
```

This plays nicely well with following [PEP 508](https://www.python.org/dev/peps/pep-0508/) dependency specification: `typing_extensions; python_version<"3.8"

### 1.9 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](https://www.python.org/dev/peps/pep-0544/) for the detailed specification of protocols and structural subtyping in Python.
## 1.9.1 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:

```python
from typing import Iterator, Iterable, Optional

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

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

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

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

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

### 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.

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

See also `Iterable`.
Iterator[T]

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

See also Iterator.

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).

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

See also Sized.

Container[T]

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

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

See also Container.

Collection[T]

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

See also Collection.

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)`.

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

See also `Reversible`.

**SupportsAbs**[T]

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

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

See also `SupportsAbs`.

**SupportsBytes**

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

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

See also `SupportsBytes`.

**SupportsComplex**

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

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

See also `SupportsComplex`.

**SupportsFloat**

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

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

See also `SupportsFloat`.

**SupportsInt**

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

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

See also `SupportsInt`. 

---

1.9. Protocols and structural subtyping 47
**SupportsRound**[T]

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

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

See also `SupportsRound`.

**Async protocols**

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

**Awaitable**[T]

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

See also `Awaitable`.

**AsyncIterable**[T]

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

See also `AsyncIterable`.

**AsyncIterator**[T]

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

See also `AsyncIterator`.

**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]

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

See also `ContextManager`. 
AsyncContextManager[T]

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

See also AsyncContextManager.

### 1.9.2 Simple user-defined protocols

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

```python
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 provided in the `typing_extensions` package for Python 3.4-3.7. Starting with Python 3.8, `Protocol` is included in the `typing` module.

### 1.9.3 Defining subprotocols and subclassing protocols

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

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

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

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

(continues on next page)
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 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.

### 1.9.4 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:

```python
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:
```

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

root: TreeLike = SimpleTree(0)  # OK

1.9.5 Using isinstance() with protocols

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

    from typing_extensions import Protocol, runtime_checkable
    @runtime_checkable
class Portable(Protocol):
        handles: int

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

    def use(handles: int) -> None: ...

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.

1.9.6 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
    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:
            ...

(continues on next page)
Callback protocols and Callable types can be used interchangeably. Argument names in __call__ methods must be identical, unless a double underscore prefix is used. For example:

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

T = TypeVar('T')

class Copy(Protocol):
    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
```

### 1.10 Dynamically typed code

As mentioned earlier, bodies of functions that don’t 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:

```python
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
```

#### 1.10.1 Operations on Any values

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

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

(continues on next page)
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:

```python
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.

### 1.10.2 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:

```python
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:

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

You can use different type narrowing techniques to narrow object to a more specific type (subtype) such as int. Type narrowing is not needed with dynamically typed values (values with type Any).

### 1.11 Type narrowing

This section is dedicated to several type narrowing techniques which are supported by mypy.

Type narrowing is when you convince a type checker that a broader type is actually more specific, for instance, that an object of type Shape is actually of the narrower type Square.

```
x.z = x(2)
open(x).read()
return x
```

1.10.2 Any vs. object
1.11.1 Type narrowing expressions

The simplest way to narrow a type is to use one of the supported expressions:

- `isinstance()` like in `isinstance(obj, float)` will narrow `obj` to have `float` type
- `issubclass()` like in `issubclass(cls, MyClass)` will narrow `cls` to be `Type[MyClass]`
- `type()` like in `type(obj) is int` will narrow `obj` to have `int` type
- `callable()` like in `callable(obj)` will narrow object to callable type

Type narrowing is contextual. For example, based on the condition, mypy will narrow an expression only within an `if` branch:

```python
def function(arg: object):
    if isinstance(arg, int):
        # Type is narrowed within the `if` branch only
        reveal_type(arg)  # Revealed type: "builtins.int"
    elif isinstance(arg, str) or isinstance(arg, bool):
        # Type is narrowed differently within this `elif` branch:
        reveal_type(arg)  # Revealed type: "builtins.str | builtins.bool"

        # Subsequent narrowing operations will narrow the type further
        if isinstance(arg, bool):
            reveal_type(arg)  # Revealed type: "builtins.bool"

    # Back outside of the `if` statement, the type isn't narrowed:
    reveal_type(arg)  # Revealed type: "builtins.object"
```

Mypy understands the implications `return` or exception raising can have for what type an object could be:

```python
def function(arg: int | str):
    if isinstance(arg, int):
        return

        # `arg` can't be `int` at this point:
        reveal_type(arg)  # Revealed type: "builtins.str"
```

We can also use `assert` to narrow types in the same context:

```python
def function(arg: Any):
    assert isinstance(arg, int)
    reveal_type(arg)  # Revealed type: "builtins.int"
```

Note: With `--warn-unreachable` narrowing types to some impossible state will be treated as an error.

```python
def function(arg: int):
    # error: Subclass of "int" and "str" cannot exist:
    # would have incompatible method signatures
    assert isinstance(arg, str)

    # error: Statement is unreachable
    print("so mypy concludes the assert will always trigger")
```
Without `--warn-unreachable` mypy will simply not check code it deems to be unreachable. See *Unreachable code* for more information.

```python
x: int = 1
assert isinstance(x, str)
reveal_type(x)  # Revealed type is "builtins.int"
print(x + '!')  # Typechecks with 'mypy', but fails in runtime.
```

### issubclass

Mypy can also use `issubclass()` for better type inference when working with types and metaclasses:

```python
class MyCalcMeta(type):
    @classmethod
def calc(cls) -> int:
        ...
def f(o: object) -> None:
t = type(o)  # We must use a variable here
reveal_type(t)  # Revealed type is "builtins.type"
    if issubclass(t, MyCalcMeta):  # `issubclass(type(o), MyCalcMeta)` won't work
        reveal_type(t)  # Revealed type is "Type[MyCalcMeta]"
t.calc()  # Okay
```

### callable

Mypy knows what types are callable and which ones are not during type checking. So, we know what `callable()` will return. For example:

```python
from typing import Callable

x: Callable[[], int]
if callable(x):
    reveal_type(x)  # N: Revealed type is "def () -> builtins.int"
else:
    ...  # Will never be executed and will raise error with `--warn-unreachable`
```

callable function can even split Union type for callable and non-callable parts:

```python
from typing import Callable, Union

x: Union[int, Callable[[], int]]
if callable(x):
    reveal_type(x)  # N: Revealed type is "def () -> builtins.int"
else:
    reveal_type(x)  # N: Revealed type is "builtins.int"
```

1.11. Type narrowing
1.11.2 Casts

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

```python
from typing import cast

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:

```python
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:

```python
from typing import cast, Any

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

1.11.3 User-Defined Type Guards

Mypy supports User-Defined Type Guards ([PEP 647](https://www.python.org/dev/peps/pep-0647/)).

A type guard is a way for programs to influence conditional type narrowing employed by a type checker based on runtime checks.

Basically, a TypeGuard is a “smart” alias for a bool type. Let’s have a look at the regular bool example:

```python
def is_str_list(val: list[object]) -> bool:
    """Determines whether all objects in the list are strings""
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]) -> None:
    if is_str_list(val):
        reveal_type(val)  # Reveals list[object]
        print(" ".join(val))  # Error: incompatible type
```

The same example with TypeGuard:
from typing import TypeGuard  # use `typing_extensions` for Python 3.9 and below

def is_str_list(val: list[object]) -> TypeGuard[list[str]]:  
    """Determines whether all objects in the list are strings""
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]) -> None:
    if is_str_list(val):
        reveal_type(val)  # list[str]
        print(" ".join(val))  # ok

How does it work? TypeGuard narrows the first function argument (val) to the type specified as the first type parameter (list[str]).

Note: Narrowing is not strict. For example, you can narrow str to int:

def f(value: str) -> TypeGuard[int]:
    return True

Note: since strict narrowing is not enforced, it's easy to break type safety.

However, there are many ways a determined or uninformed developer can subvert type safety – most commonly by using cast or Any. If a Python developer takes the time to learn about and implement user-defined type guards within their code, it is safe to assume that they are interested in type safety and will not write their type guard functions in a way that will undermine type safety or produce nonsensical results.

Generic TypeGuards

TypeGuard can also work with generic types:

from typing import TypeVar
from typing import TypeGuard  # use `typing_extensions` for `python<3.10`

_T = TypeVar("_T")

def is_two_element_tuple(val: tuple[_T, ...]) -> TypeGuard[tuple[_T, _T]]:  
    return len(val) == 2

def func(names: tuple[str, ...]):
    if is_two_element_tuple(names):
        reveal_type(names)  # tuple[str, str]
    else:
        reveal_type(names)  # tuple[str, ...]
Typeguards with parameters

Type guard functions can accept extra arguments:

```python
from typing import Type, TypeVar
from typing import TypeGuard  # use `typing_extensions` for `python<3.10`

_T = TypeVar("_T")

def is_set_of(val: set[Any], type: Type[_T]) -> TypeGuard[set[_T]]:
    return all(isinstance(x, type) for x in val)

items: set[Any]
if is_set_of(items, str):
    reveal_type(items)  # set[str]
```

TypeGuards as methods

A method can also serve as the TypeGuard:

```python
class StrValidator:
    def is_valid(self, instance: object) -> TypeGuard[str]:
        return isinstance(instance, str)

def func(to_validate: object) -> None:
    if StrValidator().is_valid(to_validate):
        reveal_type(to_validate)  # Revealed type is "builtins.str"
```

Note: Note, that TypeGuard does not narrow types of self or cls implicit arguments.

If narrowing of self or cls is required, the value can be passed as an explicit argument to a type guard function:

```python
class Parent:
    def method(self) -> None:
        reveal_type(self)  # Revealed type is "Parent"
        if is_child(self):
            reveal_type(self)  # Revealed type is "Child"

class Child(Parent):
    ...

def is_child(instance: Parent) -> TypeGuard[Child]:
    return isinstance(instance, Child)
```
Assignment expressions as TypeGuards

Sometimes you might need to create a new variable and narrow it to some specific type at the same time. This can be achieved by using TypeGuard together with := operator.

```python
from typing import TypeGuard  # use `typing_extensions` for `python<3.10`

def is_float(a: object) -> TypeGuard[float]:
    return isinstance(a, float)

def main(a: object) -> None:
    if is_float(x := a):
        reveal_type(x)  # N: Revealed type is 'builtins.float'
        reveal_type(a)  # N: Revealed type is 'builtins.object'
        reveal_type(x)  # N: Revealed type is 'builtins.object'
        reveal_type(a)  # N: Revealed type is 'builtins.object'
```

What happens here?

1. We create a new variable `x` and assign a value of `a` to it
2. We run `is_float()` type guard on `x`
3. It narrows `x` to be `float` in the `if` context and does not touch `a`

Note: The same will work with `isinstance(x := a, float)` as well.

1.12 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`.
- `bytearray` and `memoryview` are duck type compatible with `bytes`.

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:

```python
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`).
1.13 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 *Stubs files and typeshed*.

1.13.1 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.

Mypy also ships with two tools for making it easier to create and maintain stubs: *Automatic stub generation (stubgen)* and *Automatic stub testing (stubtest)*.

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.

---

1.13.2 Stub file syntax

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

If it is not possible to completely leave out some piece of runtime logic, the recommended convention is to replace or elide them with ellipsis expressions (...). Each ellipsis below is literally written in the stub file as three dots:

```python
# Variables with annotations do not need to be assigned a value.
# So by convention, we omit them in the stub file.
x: int
```

(continues on next page)
Function bodies cannot be completely removed. By convention, we replace them with `...` instead of the `pass` statement.

```python
def func_1(code: str) -> int: ...

# We can do the same with default arguments.
def func_2(a: int, b: int = ...) -> int: ...
```

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

### 1.13.3 Using stub file syntax at runtime

You may also occasionally need to elide actual logic in regular Python code – for example, when writing methods in overload variants or custom protocols.

The recommended style is to use ellipses to do so, just like in stub files. It is also considered stylistically acceptable to throw a `NotImplementedError` in cases where the user of the code may accidentally call functions with no actual logic.

You can also elide default arguments as long as the function body also contains no runtime logic: the function body only contains a single ellipsis, the pass statement, or a `raise` `NotImplementedError()`. It is also acceptable for the function body to contain a docstring. For example:

```python
from typing_extensions import Protocol

class Resource(Protocol):
    def ok_1(self, foo: list[str] = ...) -> None: ...
    def ok_2(self, foo: list[str] = ...) -> None:
        raise NotImplementedError()
    def ok_3(self, foo: list[str] = ...) -> None:
        """Some docstring""
        pass

# Error: Incompatible default for argument "foo" (default has type "ellipsis", argument has type "list[str]")
def not_ok(self, foo: list[str] = ...) -> None:
    print(foo)
```

### 1.14 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.
1.14.1 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:

```python
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:

```python
# 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:

```python
def process(stack: Stack[int]) -> None: ...
process(Stack())  # Argument has inferred type Stack[int]
```

Construction of instances of generic types is also type checked:

```python
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
```
1.14.2 Generic class internals

You may wonder what happens at runtime when you index `Stack`. Indexing `Stack` returns a *generic alias* to `Stack` that returns instances of the original class on instantiation:

```python
>>> print(Stack)
__main__.Stack

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

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

Generic aliases can be instantiated or subclassed, similar to real 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.

Note that in Python 3.8 and lower, the built-in types `list`, `dict` and others do not support indexing. This is why we have the aliases `List`, `Dict` and so on in the `typing` module. Indexing these aliases gives you a generic alias that resembles generic aliases constructed by directly indexing the target class in more recent versions of Python:

```python
>>> # Only relevant for Python 3.8 and below
>>> # For Python 3.9 onwards, prefer `list[int]` syntax
>>> from typing import List
>>> List[int]
typing.List[int]
```

Note that the generic aliases in `typing` don’t support constructing instances:

```python
>>> from typing import List
>>> List[int]()
Traceback (most recent call last):
  ...
TypeError: Type List cannot be instantiated; use list() instead
```

Note: In Python 3.6 indexing generic types or type aliases results in actual type objects. This means that generic types in type annotations can have a significant runtime cost. This was changed in Python 3.7, and indexing generic types became a cheap operation.

1.14.3 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:

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

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]:
```

(continues on next page)
...

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 f'StrDict({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[K, V] 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:

```python
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]): ...

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
```
1.14.4 Generic functions

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

```python
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:

```python
# 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:

```python
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.

1.14.5 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:

```python
from typing import TypeVar

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

class Shape:
    def set_scale(self: T, scale: float) -> T:
        (continues on next page)
```
```python
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 = Circle().set_scale(0.5).set_radius(2.7)
square: Square = Square().set_scale(0.5).set_width(3.2)
```

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]`:

```python
from typing import TypeVar, Type

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

class Friend:
    other: "Friend" = None

    @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.

For some advanced uses of self-types see additional examples.
1.14.6 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 \( \text{MyCovGen}[T, \ldots] \) is called covariant in type variable \( T \) if \( \text{MyCovGen}[B, \ldots] \) is always a subtype of \( \text{MyCovGen}[A, \ldots] \).
- A generic class \( \text{MyContraGen}[T, \ldots] \) is called contravariant in type variable \( T \) if \( \text{MyContraGen}[A, \ldots] \) is always a subtype of \( \text{MyContraGen}[B, \ldots] \).
- A generic class \( \text{MyInvGen}[T, \ldots] \) 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: \( \text{Union}[\text{Cat}, \text{int}] \) is a subtype of \( \text{Union}[\text{Animal}, \text{int}] \), \( \text{Union}[\text{Dog}, \text{int}] \) is also a subtype of \( \text{Union}[\text{Animal}, \text{int}] \), etc. Most immutable containers such as \text{Sequence} and \text{FrozenSet} are also covariant.

- **Callable** is an example of type that behaves contravariant in types of arguments, namely \( \text{Callable}[\[\text{Employee}\], \text{int}] \) is a subtype of \( \text{Callable}[\[\text{Manager}\], \text{int}] \). To understand this, consider a function:

  ```python
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:

  ```python
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 \text{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:

```python
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)
```python
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
```

### 1.14.7 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`:

```python
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:

```python
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:

```python
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:

```python
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`:

```python
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 `Pattern[AnyStr]` for the return value of `re.compile()`, since regular expressions can be based on a string or a bytes pattern.

### 1.14.8 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`.

```python
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`.

```python
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,

```python
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.

### 1.14.9 Declaring decorators

One common application of type variables along with parameter specifications is in declaring a decorator that preserves the signature of the function it decorates.

Note that class decorators are handled differently than function decorators in mypy: decorating a class does not erase its type, even if the decorator has incomplete type annotations.

Suppose we have the following decorator, not type annotated yet, that preserves the original function’s signature and merely prints the decorated function’s name:

```python
def my_decorator(func):
    def wrapper(*args, **kwd):
        print("Calling", func)
        return func(*args, **kwd)
    return wrapper
```

and we use it to decorate function `add_forty_two`:
# A decorated function.
@my_decorator
def add_forty_two(value: int) -> int:
    return value + 42

a = add_forty_two(3)

Since my_decorator is not type-annotated, the following won’t get type-checked:

```python
reveal_type(a)  # revealed type: Any
add_forty_two('foo')  # no type-checker error :(
```

Before parameter specifications, here’s how one might have annotated the decorator:

```python
from typing import Callable, TypeVar
F = TypeVar('F', bound=Callable[..., Any])

# 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)
```

and that would enable the following type checks:

```python
reveal_type(a)  # str
add_forty_two('x')  # Type check error: incompatible type "str"; expected "int"
```

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. However, with the introduction of parameter specifications in mypy 0.940, we can now have a more faithful type annotation:

```python
from typing import Callable, ParamSpec, TypeVar
P = ParamSpec('P')
T = TypeVar('T')

def my_decorator(func: Callable[P, T]) -> Callable[P, T]:
    def wrapper(*args: P.args, **kwds: P.kwargs) -> T:
        print("Calling", func)
        return func(*args, **kwds)
    return wrapper
```

When the decorator alters the signature, parameter specifications truly show their potential:

```python
from typing import Callable, ParamSpec, TypeVar
P = ParamSpec('P')
T = TypeVar('T')

# Note: We reuse 'P' in the return type, but replace 'T' with 'str'
```
def stringify(func: Callable[P, T]) -> Callable[P, str]:
    def wrapper(*args: P.args, **kwds: P.kwargs) -> str:
        return str(func(*args, **kwds))
    return wrapper

@stringify
def add_forty_two(value: int) -> int:
    return value + 42

a = add_forty_two(3)
reveal_type(a)  # str
foo('x')  # Type check error: incompatible type "str"; expected "int"

Decorator factories

Functions that take arguments and return a decorator (also called second-order decorators), are similarly supported via generics:

from typing import Any, Callable, TypeVar
F = TypeVar('F', bound=Callable[..., Any])

def route(url: str) -> Callable[[F], F]: ...

@route(url='/')
def index(request: Any) -> str:
    return 'Hello world'

Sometimes the same decorator supports both bare calls and calls with arguments. This can be achieved by combining with @overload:

from typing import Any, Callable, TypeVar, overload
F = TypeVar('F', bound=Callable[..., Any])

# Bare decorator usage
@overload
def atomic(__func: F) -> F: ...

# Decorator with arguments
@overload
def atomic(*, savepoint: bool = True) -> Callable[[F], F]: ...

# Implementation
def atomic(__func: Callable[..., Any] = None, *, savepoint: bool = True):
    def decorator(func: Callable[..., Any]):
        ...  # Code goes here
        if __func is not None:
            return decorator(__func)
        else:
            return decorator
1.14.10 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:

```python
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
```

Per [PEP 544: Generic protocols](https://www.python.org/dev/peps/pep-0544/), class `ClassName(Protocol[T])` is allowed as a shorthand for `class ClassName(Protocol, Generic[T])`.

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:

```python
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:

```python
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:

```python
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'
```

### 1.14.11 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: Type aliases):

```python
from typing import TypeVar, Iterable, Union, Callable

S = TypeVar('S')

TInt = tuple[int, S]
```

(continues on next page)
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.
1.15 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`.

### 1.15.1 The NoReturn type

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

```python
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:

```python
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
```

### 1.15.2 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:

```python
class UserId(int):
    pass

def get_by_user_id(user_id: UserId):
    ...
```
However, this approach introduces some runtime overhead. To avoid this, the typing module provides a helper object `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:

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

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

```python
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:

```python
from typing import NewType

UserId = NewType('UserId', int)

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

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

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

`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 callable 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:

```python
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()`, nor can you subclass an object returned by `NewType()`.

**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:

```python
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:

```python
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
```

### 1.15.3 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:

```python
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:

```python
from typing import Union, overload

# Overload *variants* for 'mouse_event'.
(continues on next page)
```
# 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:
        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:

```python
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
        else:
            pass
```

(continues on next page)
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.

The default values of a function’s arguments don’t affect its signature – only the absence or presence of a default value does. So in order to reduce redundancy, it’s possible to replace default values in overload definitions with ... as a placeholder:

```python
from typing import overload
class M: ...

@overload
def get_model(model_or_pk: M, flag: bool = ...) -> M: ...
@overload
def get_model(model_or_pk: int, flag: bool = ...) -> M | None: ...

def get_model(model_or_pk: int | M, flag: bool = True) -> M | None: ...
```

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.
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:

```python
# For Python 3.8 and below you must use `typing.List` instead of `list`. e.g.
# from typing import List
from typing import 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`:

```python
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:

```python
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).

**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:

```python
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:

```python
# 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:
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:

```python
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 is not 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`:

```python
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.
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`.

Conditional overloads

Sometimes it is useful to define overloads conditionally. Common use cases include types that are unavailable at runtime or that only exist in a certain Python version. All existing overload rules still apply. For example, there must be at least two overloads.

Note: Mypy can only infer a limited number of conditions. Supported ones currently include `TYPE_CHECKING`, `MYPY`, `Python version and system platform checks`, `--always-true`, and `--always-false` values.

```python
from typing import TYPE_CHECKING, Any, overload

if TYPE_CHECKING:
    class A: ...
    class B: ...

if TYPE_CHECKING:
    @overload
    def func(var: A) -> A: ...

    @overload
    def func(var: B) -> B: ...

def func(var: Any) -> Any:
    return var

reveal_type(func(A()))  # Revealed type is "A"
```
# flags: --python-version 3.10
import sys
from typing import Any, overload

class A: ...
class B: ...
class C: ...
class D: ...

if sys.version_info < (3, 7):
    @overload
def func(var: A) -> A: ...

elif sys.version_info >= (3, 10):
    @overload
def func(var: B) -> B: ...

else:
    @overload
def func(var: C) -> C: ...

@overload
def func(var: D) -> D: ...

def func(var: Any) -> Any:
    return var

reveal_type(func(B()))  # Revealed type is "B"
reveal_type(func(C()))  # No overload variant of "func" matches argument type "C"
    # Possible overload variants:
    #    def func(var: B) -> B
    #    def func(var: D) -> D
    # Revealed type is "Any"

Note: In the last example, mypy is executed with --python-version 3.10. Therefore, the condition sys.version_info >= (3, 10) will match and the overload for B will be added. The overloads for A and C are ignored! The overload for D is not defined conditionally and thus is also added.

When mypy cannot infer a condition to be always True or always False, an error is emitted.
def func(var: A) -> A: ...

@overload
def func(var: B) -> B: ...

def func(var: Any) -> Any: ...

reveal_type(func(A()))  # Revealed type is "Any"

1.15.4 Advanced uses of self-types

Normally, mypy doesn’t require annotations for the first arguments of instance and class methods. However, they may be needed to have more precise static typing for certain programming patterns.

Restricted methods in generic classes

In generic classes some methods may be allowed to be called only for certain values of type arguments:

```python
T = TypeVar('T')

class Tag(Generic[T]):
    item: T
    def uppercase_item(self: Tag[str]) -> str:
        return self.item.upper()

def label(ti: Tag[int], ts: Tag[str]) -> None:
    ti.uppercase_item()  # E: Invalid self argument "Tag[int]" to attribute function
    # "uppercase_item" with type "Callable[[Tag[str]], str]"
    ts.uppercase_item()  # This is OK
```

This pattern also allows matching on nested types in situations where the type argument is itself generic:

```python
T = TypeVar('T', covariant=True)
S = TypeVar('S')

class Storage(Generic[T]):
    def __init__(self, content: T) -> None:
        self.content = content
    def first_chunk(self: Storage[Sequence[S]]) -> S:
        return self.content[0]

page: Storage[list[str]]
page.first_chunk()  # OK, type is "str"

Storage(0).first_chunk()  # Error: Invalid self argument "Storage[int]" to attribute
# "first_chunk" with type "Callable[[Storage[Sequence[S]]], S]"
```

Finally, one can use overloads on self-type to express precise types of some tricky methods:

```python
1.15. More types 85
```
T = TypeVar('T')

class Tag(Generic[T]):
    @overload
    def export(self: Tag[str]) -> str: ...
    @overload
    def export(self, converter: Callable[[T], str]) -> str: ...

    def export(self, converter=None):
        if isinstance(self.item, str):
            return self.item
        return converter(self.item)

In particular, an \_\_init\_\_\() method overloaded on self-type may be useful to annotate generic class constructors where type arguments depend on constructor parameters in a non-trivial way, see e.g. \_\_Popen\_.

**Mixin classes**

Using host class protocol as a self-type in mixin methods allows more code re-usability for static typing of mixin classes. For example, one can define a protocol that defines common functionality for host classes instead of adding required abstract methods to every mixin:

class Lockable(Protocol):
    @property
    def lock(self) -> Lock: ...

class AtomicCloseMixin:
    def atomic_close(self: Lockable) -> int:
        with self.lock:
            # perform actions

class AtomicOpenMixin:
    def atomic_open(self: Lockable) -> int:
        with self.lock:
            # perform actions

class File(AtomicCloseMixin, AtomicOpenMixin):
    def \_\_init\_\_\_(self) -> None:
        self.lock = Lock()

class Bad(AtomicCloseMixin):
    pass

f = File()
b: Bad
f.atomic_close()  # OK
b.atomic_close()  # Error: Invalid self type for "atomic_close"

Note that the explicit self-type is \texttt{required} to be a protocol whenever it is not a supertype of the current class. In this case mypy will check the validity of the self-type only at the call site.
Precise typing of alternative constructors

Some classes may define alternative constructors. If these classes are generic, self-type allows giving them precise signatures:

\[
\begin{align*}
\text{T} &= \text{TypeVar('T')} \\
\text{class Base(Generic[T]):} \\
\text{~~Q &= \text{TypeVar('Q', bound='Base[T]')} } \\
\text{~~def \_init\_}(self, item: T) -> \text{None}: \\
\text{~~~~self.item = item} \\
\text{~~@classmethod} \\
\text{~~def make_pair}(cls: \text{Type}[Q], item: T) -> \text{tuple}[Q, Q]: \\
\text{~~~~return cls(item), cls(item)} \\
\text{class Sub(Base[T]):} \\
\text{~~...} \\
\text{pair &= Sub.make_pair('yes')} \quad \# \text{Type is } \text{"tuple[Sub[\text{str}], Sub[\text{str}]\"]}\n\text{bad &= Sub[int].make_pair('no')} \quad \# \text{Error: Argument 1 to } \text{"make_pair" of } \text{"Base"} \\
\text{~~~~has incompatible type } \text{"str"; expected } \text{"int"}
\end{align*}
\]

1.15.5 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.

```python
import asyncio

async def format_string(tag: str, count: int) -> str:
    return f"T-minus {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 Coroutine[\text{Any, Any, T}], which is a subtype of Awaitable[T]:

1.15. More types

87
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.

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

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

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

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

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

If you use coroutines in legacy code that was originally written for Python 3.4, which did not support the `async def` syntax, you would instead use the `@asyncio.coroutine` decorator to convert a generator into a coroutine, and use a generator type as the return type:

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

@asyncio.coroutine
def countdown_2(tag: str, count: int) -> Generator[Any, None, str]:
    while count > 0:
        print(f'T-minus {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()
```

### 1.15.6 TypedDict

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

```python
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:

```python
from typing_extensions import TypedDict
Movie = TypedDict('Movie', {'name': str, 'year': int})
movie: Movie = {'name': 'Blade Runner', 'year': 1982}
```

`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:

```python
define 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:

```python
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:

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

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

Like all types, `TypedDicts` can be used as components to build arbitrarily complex types. For example, you can define nested `TypedDicts` and containers with `TypedDict` items. Unlike most other types, `mypy` uses structural compatibility checking (or structural subtyping) with `TypedDicts`. A `TypedDict` object with extra items is compatible with (a 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 `Mapping` only provides read-only access to the dictionary items:

```python
def print_typed_dict(obj: Mapping[str, object]) -> None:
    for key, value in obj.items():
        print(f'{key}: {value}')

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

**Note:** Unless you are on Python 3.8 or newer (where `TypedDict` is available in standard library `typing` module) you need to install `typing_extensions` using `pip` to use `TypedDict`:

```bash
code python3 -m pip install --upgrade typing-extensions
```
**Totality**

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

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

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:

```python
GuiOptions = TypedDict(
    'GuiOptions', {'language': str, 'color': str}, total=False
)
options: 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:

```python
# 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.

**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 `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)
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.

**Class-based syntax**

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

```python
from typing_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 `TypedDict`s, inheritance is not required for compatibility. Here is an example of inheritance:

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

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

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

**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:

```python
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`. 
Unions of TypedDicts

Since TypedDicts are really just regular dicts at runtime, it is not possible to use `isinstance` checks to distinguish between different variants of a Union of TypedDict in the same way you can with regular objects.

Instead, you can use the tagged union pattern. The referenced section of the docs has a full description with an example, but in short, you will need to give each TypedDict the same key where each value has a unique Literal type. Then, check that key to distinguish between your TypedDicts.

1.16 Literal types and Enums

1.16.1 Literal types

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:

```python
from typing import overload, Union, 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
    ...
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]"
```

Note: The examples in this page import `Literal` as well as `Final` and `TypedDict` from the `typing` module. These types were added to `typing` in Python 3.8, but are also available for use in Python 3.4 - 3.7 via the `typing_extensions`
Parameterizing Literals

Literal types may contain one or more literal bools, ints, strs, bytes, and enum values. 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", MyEnum.A]` is equivalent to `Union[Literal[-3], Literal[b"foo"], Literal[MyEnum.A]]`. 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:

```python
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.

Declaring literal variables

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

```python
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:

```python
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` (see Final names, methods and classes):

```python
from typing import Final, Literal

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

c: Final = 19
reveal_type(c) # Revealed type is "Literal[19]"?
events_literal(c) # ...and 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. This is why the revealed type of `c` is `Literal[19]?`: the question mark at the end reflects this context-sensitive nature.

For example, mypy will type check the above program almost as if it were written like so:

```python
from typing import Final, Literal
def expects_literal(x: Literal[19]) -> None: pass
reveal_type(19)
equals_literal(19)
```

This means that 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.

The main cases where the behavior of context-sensitive vs true literal types differ are when you try using those types in places that are not explicitly expecting a `Literal[...]`. For example, compare and contrast what happens when you try appending these types to a list:

```python
from typing import Final, Literal
a: Final = 19
b: Literal[19] = 19
# Mypy will choose to infer list[int] here.
list_of_ints = []
list_of_ints.append(a)
reveal_type(list_of_ints)  # Revealed type is "list[int]"

# But if the variable you're appending is an explicit Literal, mypy
# will infer list[Literal[19]].
list_of_lits = []
list_of_lits.append(b)
reveal_type(list_of_lits)  # Revealed type is "list[Literal[19]]"
```

### Intelligent indexing

We can use `Literal` types to more precisely index into structured heterogeneous types such as tuples, NamedTuples, and `TypedDicts`. This feature is known as *intelligent indexing*.

For example, when we index into a tuple using some int, the inferred type is normally the union of the tuple item types. However, if we want just the type corresponding to some particular index, we can use `Literal` types like so:

```python
from typing import TypedDict
tup = ("foo", 3.4)
# Indexing with an int literal gives us the exact type for that index
reveal_type(tup[0])  # Revealed type is "str"

# But what if we want the index to be a variable? Normally mypy won't
# know exactly what the index is and so will return a less precise type:
int_index = 0
```

(continues on next page)
reveal_type(tup[int_index])  # Revealed type is "Union[str, float]"

# But if we use either Literal types or a Final int, we can gain back
# the precision we originally had:
lit_index: Literal[0] = 0
fin_index: Final = 0
reveal_type(tup[lit_index])  # Revealed type is "str"
reveal_type(tup[fin_index])  # Revealed type is "str"

# We can do the same thing with with TypedDict and str keys:
class MyDict(TypedDict):
    name: str
    main_id: int
    backup_id: int
d: MyDict = {"name": "Saanvi", "main_id": 111, "backup_id": 222}
name_key: Final = "name"
reveal_type(d[name_key])  # Revealed type is "str"

id_key: Literal["main_id", "backup_id"]
reveal_type(d[id_key])  # Revealed type is "int"

Tagged unions

When you have a union of types, you can normally discriminate between each type in the union by using `isinstance` checks. For example, if you had a variable `x` of type `Union[int, str]`, you could write some code that runs only if `x` is an int by doing `if isinstance(x, int): ....`

However, it is not always possible or convenient to do this. For example, it is not possible to use `isinstance` to distinguish between two different `TypedDict` since at runtime, your variable will simply be just a dict.

Instead, what you can do is *label* or *tag* your `TypedDicts` with a distinct `Literal` type. Then, you can discriminate between each kind of `TypedDict` by checking the label:

```python
from typing import Literal, TypedDict, Union

class NewJobEvent(TypedDict):
    tag: Literal["new-job"]
    job_name: str
    config_file_path: str

class CancelJobEvent(TypedDict):
    tag: Literal["cancel-job"]
    job_id: int

Event = Union[NewJobEvent, CancelJobEvent]

def process_event(event: Event) -> None:
    # Since we made sure both `TypedDicts` have a key named 'tag', it's
    # safe to do 'event["tag"]'. This expression normally has the type
    # Literal["new-job", "cancel-job"], but the check below will narrow
```
# the type to either `Literal["new-job"]` or `Literal["cancel-job"]`

# This in turns narrows the type of `event` to either `NewJobEvent` or `CancelJobEvent`

```python
if event["tag"] == "new-job":
    print(event["job_name"])
else:
    print(event["job_id"])
```

While this feature is mostly useful when working with TypedDicts, you can also use the same technique with regular objects, tuples, or namedtuples.

Similarly, tags do not need to be specifically str Literals: they can be any type you can normally narrow within if statements and the like. For example, you could have your tags be int or Enum Literals or even regular classes you narrow using `isinstance()`:

```python
from typing import Generic, TypeVar, Union

T = TypeVar('T')

class Wrapper(Generic[T]):
    def __init__(self, inner: T) -> None:
        self.inner = inner

    def process(w: Union[Wrapper[int], Wrapper[str]]) -> None:
        # Doing `if isinstance(w, Wrapper[int])` does not work: `isinstance` requires
        # that the second argument always be an "erased" type, with no generics.
        # This is because generics are a typing-only concept and do not exist at
        # runtime in a way `isinstance` can always check.
        #
        # However, we can side-step this by checking the type of `w.inner` to
        # narrow `w` itself:
        if isinstance(w.inner, int):
            reveal_type(w) # Revealed type is "Wrapper[int]"
        else:
            reveal_type(w) # Revealed type is "Wrapper[str]"
```

This feature is sometimes called “sum types” or “discriminated union types” in other programming languages.

**Exhaustiveness checking**

You may want to check that some code covers all possible Literal or Enum cases. Example:

```python
from typing import Literal

PossibleValues = Literal['one', 'two']

def validate(x: PossibleValues) -> bool:
    if x == 'one':
        return True
    elif x == 'two':
        return False
```

1.16. Literal types and Enums
In the code above, it's easy to make a mistake. You can add a new literal value to PossibleValues but forget to handle it in the validate function:

```
PossibleValues = Literal['one', 'two', 'three']

Mypy won't catch that 'three' is not covered. If you want mypy to perform an exhaustiveness check, you need to update your code to use an `assert_never()` check:

```
from typing import Literal, NoReturn

PossibleValues = Literal['one', 'two']

def assert_never(value: NoReturn) -> NoReturn:
    # This also works at runtime as well
    assert False, f'This code should never be reached, got: {value}'

def validate(x: PossibleValues) -> bool:
    if x == 'one':
        return True
    elif x == 'two':
        return False
    assert_never(x)
```

Now if you add a new value to PossibleValues but don't update validate, mypy will spot the error:

```
PossibleValues = Literal['one', 'two', 'three']

def validate(x: PossibleValues) -> bool:
    if x == 'one':
        return True
    elif x == 'two':
        return False
    # Error: Argument 1 to "assert_never" has incompatible type "Literal['three']";
    # expected "NoReturn"
    assert_never(x)
```

If runtime checking against unexpected values is not needed, you can leave out the `assert_never` call in the above example, and mypy will still generate an error about function validate returning without a value:

```
PossibleValues = Literal['one', 'two', 'three']

# Error: Missing return statement
def validate(x: PossibleValues) -> bool:
    if x == 'one':
        return True
    elif x == 'two':
        return False
```

Exhaustiveness checking is also supported for match statements (Python 3.10 and later):
```python
def validate(x: PossibleValues) -> bool:
    match x:
        case 'one':
            return True
        case 'two':
            return False
    assert_never(x)
```

### 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__`.

#### 1.16.2 Enums


```python
from enum import Enum

class Direction(Enum):
    up = 'up'
    down = 'down'

reveal_type(Direction.up)  # Revealed type is "Literal[Direction.up]?"
reveal_type(Direction.down)  # Revealed type is "Literal[Direction.down]?
```

You can use enums to annotate types as you would expect:

```python
class Movement:
    def __init__(self, direction: Direction, speed: float) -> None:
        self.direction = direction
        self.speed = speed

Movement(Direction.up, 5.0)  # ok
Movement('up', 5.0)  # E: Argument 1 to "Movement" has incompatible type "str"; expected "Direction"
```
Exhaustiveness checking

Similar to Literal types, Enum supports exhaustiveness checking. Let’s start with a definition:

```python
from enum import Enum
from typing import NoReturn

def assert_never(value: NoReturn) -> NoReturn:
    # This also works in runtime as well:
    assert False, f'This code should never be reached, got: {value}'

class Direction(Enum):
    up = 'up'
    down = 'down'
```

Now, let’s use an exhaustiveness check:

```python
def choose_direction(direction: Direction) -> None:
    if direction is Direction.up:
        reveal_type(direction) # N: Revealed type is "Literal[Direction.up]"
        print('Going up!')
        return
    elif direction is Direction.down:
        print('Down')
        return
    # This line is never reached
    assert_never(direction)
```

If we forget to handle one of the cases, mypy will generate an error:

```python
def choose_direction(direction: Direction) -> None:
    if direction == Direction.up:
        print('Going up!')
        return
    assert_never(direction) # E: Argument 1 to "assert_never" has incompatible type ->"Direction"; expected "NoReturn"
```

Exhaustiveness checking is also supported for match statements (Python 3.10 and later).

Extra Enum checks

Mypy also tries to support special features of Enum the same way Python’s runtime does:

- Any Enum class with values is implicitly final. This is what happens in CPython:

```python
>>> class AllDirection(Direction):
...     left = 'left'
...     right = 'right'
Traceback (most recent call last):
  ... TypeError: Other: cannot extend enumeration 'Some'
```

Mypy also catches this error:
```python
class AllDirection(Direction):  # E: Cannot inherit from final class "Some"
    left = 'left'
    right = 'right'
```

- All Enum fields are implicitly final as well.

```python
Direction.up = '^'  # E: Cannot assign to final attribute "up"
```

- All field names are checked to be unique.

```python
class Some(Enum):
    x = 1
    x = 2  # E: Attempted to reuse member name "x" in Enum definition "Some"
```

- Base classes have no conflicts and mixin types are correct.

```python
class WrongEnum(str, int, enum.Enum):
    # E: Only a single data type mixin is allowed for Enum subtypes, found extra
    → "int"
    ...

class MixinAfterEnum(enum.Enum, Mixin):
    # E: No base classes are allowed after
    → "enum.Enum"
    ...
```

### 1.17 Final names, methods and classes

This section introduces these related features:

1. **Final names** are variables or attributes that should not be 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. There is no runtime enforcement by the Python runtime.

**Note:** The examples in this page import `Final` and `final` from the `typing` module. These types were added to `typing` in Python 3.8, but are also available for use in Python 3.4 - 3.7 via the `typing_extensions` package.

#### 1.17.1 Final names

You can use the `typing.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:

```python
from typing import Final

RATE: Final = 3_000
```
Another use case for final attributes is to protect certain attributes from being overridden in a subclass:

```python
from typing 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.

**Syntax variants**

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

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

  ```python
  ID: Final[int] = 1
  ```

  Here mypy will infer type `int` for `ID`.

- You can omit the type:

  ```python
  ID: Final = 1
  ```

  Here mypy will infer type `Literal[1]` 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[int]`.

- 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.

**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):
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:

```python
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:

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

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

Declaring a name as final only guarantees that the name will 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:

```python
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
```

1.17.2 Final methods

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

```python
from typing 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):

```python
from typing import Any, overload

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

### 1.17.3 Final classes

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

```python
from typing 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.

An abstract class that defines at least one abstract method or property and has @final decorator will generate an error from mypy, since those attributes could never be implemented.

```python
from abc import ABCMeta, abstractmethod
from typing import final

@final
class A(metaclass=ABCMeta):  # error: Final class A has abstract attributes "f"
    @abstractmethod
def f(self, x: int) -> None: pass
```
1.18 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`.

1.18.1 Defining a metaclass

```python
class M(type):
    pass

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

1.18.2 Metaclass usage example

Mypy supports the lookup of attributes in the metaclass:

```python
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", "int" and "str")
```
1.18.3 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:

```python
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.

1.19 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.

1.19.1 Specifying code to be checked

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

Note that if you use namespace packages (in particular, packages without `__init__.py`), you’ll need to specify `--namespace-packages`.

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).

### 1.19.2 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.
1.19.3 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 filesystem. 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.

1.19.4 Missing imports

When you import a module, mypy may report that it is unable to follow the import. This can cause errors that look like the following:

```
main.py:1: error: Skipping analyzing 'django': module is installed, but missing library stubs or py.typed marker
main.py:2: error: Library stubs not installed for "requests" (or incompatible with Python 3.8)
main.py:3: error: Cannot find implementation or library stub for module named "this_module_does_not_exist"
```

If you get any of these errors on an import, mypy will assume the type of that module is `Any`, the dynamic type. This means attempting to access any attribute of the module will automatically succeed:

```python
# Error: Cannot find implementation or library stub for module named 'does_not_exist'
import does_not_exist

# But this type checks, and x will have type 'Any'
x = does_not_exist.foobar()
```

The next sections describe what each of these errors means and recommended next steps; scroll to the section that matches your error.

**Missing library stubs or py.typed marker**

If you are getting a Skipping analyzing X: module is installed, but missing library stubs or py.typed marker error, this means mypy was able to find the module you were importing, but no corresponding type hints.

Mypy will not try inferring the types of any 3rd party libraries you have installed unless they either have declared themselves to be PEP 561 compliant stub package (e.g. with a `py.typed` file) or have registered themselves on typeshed, the repository of types for the standard library and some 3rd party libraries.

If you are getting this error, try:

1. Upgrading the version of the library you're using, in case a newer version has started to include type hints.
2. 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.
For example, if you want type hints for the `django` library, you can install the `django-stubs` package.

3. **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 using the `files` or `mypy_path` config file options, or by adding the location to the `MYPYPATH` environment variable.

These stub files do not need to be complete! A good strategy is to use stubgen, a program that comes bundled with mypy, to generate a first rough draft of the stubs. You can then iterate on just the parts of the library you need.

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 you are unable to find any existing type hints nor have time to write your own, you can instead suppress the errors. All this will do is make mypy stop reporting an error on the line containing the import: the imported module will continue to be of type `Any`.

1. To suppress a single missing import error, add a `# type: ignore` at the end of the line containing the import.

2. To suppress all missing import 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. The `.*` after `foobar` will ignore imports of `foobar` modules and subpackages in addition to the `foobar` top-level package namespace.

3. To suppress 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.

**Library stubs not installed**

If mypy can’t find stubs for a third-party library, and it knows that stubs exist for the library, you will get a message like this:

```
main.py:1: error: Library stubs not installed for "yaml" (or incompatible with Python 3.8)
main.py:1: note: Hint: "python3 -m pip install types-PyYAML"
main.py:1: note: (or run "mypy --install-types" to install all missing stub packages)
```

You can resolve the issue by running the suggested pip command or commands. Alternatively, you can use `--install-types` to install all known missing stubs:

```
mypy --install-types
```
This installs any stub packages that were suggested in the previous mypy run. You can also use your normal mypy command line with the extra `--install-types` option to install missing stubs at the end of the run (if any were found).

Use `--install-types` with `--non-interactive` to install all suggested stub packages without asking for confirmation, and type check your code, in a single command:

```
mypy --install-types --non-interactive src/
```

This can be useful in Continuous Integration jobs if you’d prefer not to manage stub packages manually. This is somewhat slower than explicitly installing stubs before running mypy, since it may type check your code twice – the first time to find the missing stubs, and the second time to type check your code properly after mypy has installed the stubs.

### Cannot find implementation or library stub

If you are getting a `Cannot find implementation or library stub for module` error, this means mypy was not able to find the module you are trying to import, whether it comes bundled with type hints or not. If you are getting this error, try:

1. Making sure your import does not contain a typo.
2. If the module is a third party library, making sure that mypy is able to find the interpreter containing the installed library.
   
   For example, if you are running your code in a virtualenv, make sure to install and use mypy within the virtualenv. Alternatively, if you want to use a globally installed mypy, set the `--python-executable` command line flag to point the Python interpreter containing your installed third party packages.
3. 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.
4. Directly specifying the directory containing the module you want to type check from the command line, by using the `mypy_path` or `files` config file options, or by using the `MYPYPATH` environment variable.
   
   Note: if the module you are trying to import is actually a submodule of some package, you should specify the directory containing the entire package. 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 must run `mypy ~/foo-project/src` (or set the `MYPYPATH` to `~/foo-project/src`.
5. If you are using namespace packages – packages which do not contain `__init__.py` files within each subfolder – using the `--namespace-packages` command line flag.

In some rare cases, you may get the “Cannot find implementation or library stub for module” error even when the module is installed in your system. This can happen when the module is both missing type hints and is installed on your system in an unconventional way.

In this case, follow the steps above on how to handle missing type hints in third party libraries.
1.19.5 Following imports

Mypy is designed to \textit{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 \texttt{mypy -m mycode.foo} and mypy discovers that mycode.foo imports mycode.bar.

How do we want mypy to type check mycode.bar? Mypy’s behaviour here is configurable -- although we \textbf{strongly recommend} using the default -- by using the \textit{--follow-imports} flag. This flag accepts one of four string values:

- \texttt{normal} (the default, recommended) 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).
- \texttt{silent} behaves in the same way as \texttt{normal} but will additionally \texttt{suppress} any error messages.
- \texttt{skip} will \texttt{not} follow imports and instead will silently replace the module (and \textit{anything imported from it}) with an object of type \texttt{Any}.
- \texttt{error} behaves in the same way as \texttt{skip} but is not quite as silent -- it will flag the import as an error, like this:

\begin{verbatim}
main.py:1: note: Import of "mycode.bar" ignored
main.py:1: note: (Using --follow-imports=error, module not passed on command line)
\end{verbatim}

If you are starting a new codebase and plan on using type hints from the start, we recommend you use either \texttt{--follow-imports=normal} (the default) or \texttt{--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 \texttt{--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.

Only if doing this is intractable, we recommend passing mypy just the files you want to type check and use \texttt{--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 \textit{Using mypy with an existing codebase} for more recommendations.

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

Adjusting import following behaviour is often most useful when restricted to specific modules. This can be accomplished by setting a per-module \texttt{follow_imports} config option.

1.19.6 Mapping file paths to modules

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

\begin{verbatim}
$ mypy file_1.py foo/file_2.py file_3.pyi some/directory
\end{verbatim}

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

- Mypy will check all paths provided that correspond to files.
- Mypy will recursively discover and check all files ending in .py or .pyi in directory paths provided, after accounting for \texttt{--exclude}.
- For each file to be checked, mypy will attempt to associate the file (e.g. \texttt{project/foo/bar/baz.py}) with a fully qualified module name (e.g. \texttt{foo.bar.baz}). The directory the package is in (\texttt{project}) is then added to mypy’s module search paths.
How mypy determines fully qualified module names depends on if the options `--namespace-packages` and `--explicit-package-bases` are set.

1. If `--namespace-packages` is off, mypy will rely solely upon the presence of `__init__.py` files to determine the fully qualified module name. That is, mypy will crawl up the directory tree for as long as it continues to find `__init__.py` (or `__init__.pyi`) files.

   For example, if your directory tree consists of `pkg/subpkg/mod.py`, mypy would require `pkg/__init__.py` and `pkg/subpkg/__init__.py` to exist in order correctly associate `mod.py` with `pkg.subpkg.mod`.

2. If `--namespace-packages` is on, but `--explicit-package-bases` is off, mypy will allow for the possibility that directories without `__init__.py` are packages. Specifically, mypy will look at all parent directories of the file and use the location of the highest `__init__.py` in the directory tree to determine the top-level package.

   For example, say your directory tree consists solely of `pkg/__init__.py` and `pkg/a/b/c/d/mod.py`. When determining `mod.py`’s fully qualified module name, mypy will look at `pkg/__init__.py` and conclude that the associated module name is `pkg.a.b.c.d.mod`.

3. You'll notice that the above case still relies on `__init__.py`. If you can't put an `__init__.py` in your top-level package, but still wish to pass paths (as opposed to packages or modules using the `-p` or `-m` flags), `--explicit-package-bases` provides a solution.

   With `--explicit-package-bases`, mypy will locate the nearest parent directory that is a member of the `MYPYPATH` environment variable, the `mypy_path` config or is the current working directory. Mypy will then use the relative path to determine the fully qualified module name.

   For example, say your directory tree consists solely of `src/namespace_pkg/mod.py`. If you run the following command, mypy will correctly associate `mod.py` with `namespace_pkg.mod`:

   ```
   $ MYPYPATH=src mypy --namespace-packages --explicit-package-bases .
   ```

   If you pass a file not ending in `.py` i.e., the module name assumed is `__main__` (matching the behavior of the Python interpreter), unless `--scripts-are-modules` is passed.

   Passing `--` will show you the files and associated module names that mypy will check.

### 1.19.7 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 `Mapping file paths to modules`).
- 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 stub-only package (PEP 561) via the `MYPYPATH`, it must be installed (see `PEP 561 support`).
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.)

### 1.19.8 Other advice and best practices

There are multiple ways of telling mypy what files to type check, ranging from passing in command line arguments to using the `files` or `mypy_path` config file options to setting the `MYPYPATH` environment variable.

However, in practice, it is usually sufficient to just use either command line arguments or the `files` config file option (the two are largely interchangeable).

Setting `mypy_path`/`MYPYPATH` is mostly useful in the case where you want to try running mypy against multiple distinct sets of files that happen to share some common dependencies.

For example, if you have multiple projects that happen to be using the same set of work-in-progress stubs, it could be convenient to just have your `MYPYPATH` point to a single directory containing the stubs.

### 1.20 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.

#### 1.20.1 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.

--exclude
A regular expression that matches file names, directory names and paths which mypy should ignore while recursively discovering files to check. Use forward slashes on all platforms.
For instance, to avoid discovering any files named setup.py you could pass --exclude '/setup\..py$'. Similarly, you can ignore discovering directories with a given name by e.g. --exclude /build/ or those matching a subpath with --exclude /project/vendor/. To ignore multiple files / directories / paths, you can provide the --exclude flag more than once, e.g --exclude '/setup\.py$' --exclude '/build/'.
Note that this flag only affects recursive directory tree discovery, that is, when mypy is discovering files within a directory tree or submodules of a package to check. If you pass a file or module explicitly it will still be checked.
For instance, mypy --exclude '/setup.py$' but_still_check/setup.py.
In particular, --exclude does not affect mypy's import following. You can use a per-module follow_imports config option to additionally avoid mypy from following imports and checking code you do not wish to be checked.
Note that mypy will never recursively discover files and directories named “site-packages”, “node_modules” or “__pycache__”, or those whose name starts with a period, exactly as --exclude '/(site-packages|node_modules|__pycache__|\..*)/$' would. Mypy will also never recursively discover files with extensions other than .py or .pyi.

1.20.2 Optional arguments

-h, --help
Show help message and exit.

-v, --verbose
More verbose messages.

-V, --version
Show program’s version number and exit.

1.20.3 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, .mypy.ini, pyproject.toml, or setup.cfg in the current directory. Settings override mypy’s built-in defaults and command line flags can override settings.
Specifying --config-file= (with no filename) will ignore all config files.
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. (This requires turning off incremental mode using --no-incremental.)
1.20.4 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.

This flag affects how mypy finds modules and packages explicitly passed on the command line. It also affects how mypy determines fully qualified module names for files passed on the command line. See Mapping file paths to modules for details.

--explicit-package-bases
This flag tells mypy that top-level packages will be based in either the current directory, or a member of the MYPYPATH environment variable or mypy_path config option. This option is only useful in conjunction with --namespace-packages. See Mapping file paths to modules for details.

--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:

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

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

```python
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.

--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.

`--fast-module-lookup`

The default logic used to scan through search paths to resolve imports has a quadratic worse-case behavior in some cases, which is for instance triggered by a large number of folders sharing a top-level namespace as in:

```
foo/
    company/
    foo/
        a.py

bar/
    company/
    bar/
        b.py

baz/
    company/
    baz/
        c.py
...
```

If you are in this situation, you can enable an experimental fast path by setting the `--fast-module-lookup` option.

### 1.20.5 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.

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).

`--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.
1.20.6 Disallow dynamic typing

The Any type is used to 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. For example, you can’t use a bare x: list. Instead, you must always write something like x: list[int].

--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.

1.20.7 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.

1.20.8 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:

```python
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]:

```python
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).

1.20.9 Configuring warnings

The following 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.

`--warn-unreachable`
This flag will make mypy report an error whenever it encounters code determined to be unreachable or redundant after performing type analysis. This can be a helpful way of detecting certain kinds of bugs in your code.

For example, enabling this flag will make mypy report that the `x > 7` check is redundant and that the `else` block below is unreachable.

```python
def process(x: int) -> None:
    # Error: Right operand of "or" is never evaluated
    if isinstance(x, int) or x > 7:
        # Error: Unsupported operand types for + ("int" and "str")
        print(x + "bad")
    else:
        # Error: 'Statement is unreachable' error
        print(x + "bad")
```

To help prevent mypy from generating spurious warnings, the “Statement is unreachable” warning will be silenced in exactly two cases:

1. When the unreachable statement is a `raise` statement, an `assert False` statement, or calls a function that has the `NoReturn` return type hint. In other words, when the unreachable statement throws an error or terminates the program in some way.

2. When the unreachable statement was intentionally marked as unreachable using Python version and system platform checks.

**Note:** Mypy currently cannot detect and report unreachable or redundant code inside any functions using Type variables with value restriction.

This limitation will be removed in future releases of mypy.

### 1.20.10 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 redefinition 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]
    # 'items' now has type list[list[str]]

    The variable must be used before it can be redefined:

def process(items: list[str]) -> None:
    items = "mypy"  # invalid redefinition to str because the variable hasn't been used yet
    print(items)
    items = "100"  # valid, items now has type str
    items = int(items)  # valid, items now has type int

--local-partial-types

In mypy, the most common cases for partial types are variables initialized using None, but without explicit Optional annotations. By default, mypy won’t check partial types spanning module top level or class top level. This flag changes the behavior to only allow partial types at local level, therefore it disallows inferring variable type for None from two assignments in different scopes. For example:

from typing import Optional

a = None  # Need type annotation here if using --local-partial-types
b: Optional[int] = None

class Foo:
    bar = None  # Need type annotation here if using --local-partial-types
    baz: Optional[int] = None

    def __init__(self) -> None:
        self.bar = 1

reveal_type(Foo().bar)  # Union[int, None] without --local-partial-types

Note: this option is always implicitly enabled in mypy daemon and will become enabled by default for mypy in a future release.

--no-implicit-reexport

By default, imported values to a module are treated as exported and mypy allows other modules to import them. This flag changes the behavior to not re-export unless the item is imported using from-as or is included in __all__. Note this is always treated as enabled for stub files. For example:

# This won't re-export the value
from foo import bar

# Neither will this
from foo import bar as bang

# This will re-export it as bar and allow other modules to import it
from foo import bar as bar

# This will also re-export bar
from foo import bar
__all__ = ['bar']
--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:

```python
from typing import Text

items: list[int]
if 'some string' in items:  # Error: non-overlapping container check!
    ...

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

assert text is not None  # OK, check against None is allowed as a special case.
```

--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.

--disable-error-code
This flag allows disabling one or multiple error codes globally. See Error codes for more information.

```python
# no flag
x = 'a string'
x.trim()  # error: "str" has no attribute "trim" [attr-defined]

# When using --disable-error-code attr-defined
x = 'a string'
x.trim()
```

--enable-error-code
This flag allows enabling one or multiple error codes globally. See Error codes for more information.

Note: This flag will override disabled error codes from the `--disable-error-code` flag.

```python
# When using --disable-error-code attr-defined
x = 'a string'
x.trim()

# --disable-error-code attr-defined --enable-error-code attr-defined
x = 'a string'
x.trim()  # error: "str" has no attribute "trim" [attr-defined]
```
1.20.11 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:

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

--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")
```

--show-error-end
This flag will make mypy show not just that start position where an error was detected, but also the end position of the relevant expression. This way various tools can easily highlight the whole error span. The format is file:line:column:end_line:end_column. This option implies --show-column-numbers.

--show-error-codes
This flag will add an error code [<code>] to error messages. The error code is shown after each error message:

```
prog.py:1: error: "str" has no attribute "trim" [attr-defined]
```

See Error codes for more information.

--pretty
Use visually nicer output in error messages: use soft word wrap, show source code snippets, and show error location markers.

--no-color-output
This flag will disable color output in error messages, enabled by default.

--no-error-summary
This flag will disable error summary. By default mypy shows a summary line including total number of errors, number of files with errors, and number of files checked.

--show-absolute-path
Show absolute paths to files.

--soft-error-limit N
This flag will adjust the limit after which mypy will (sometimes) disable reporting most additional errors. The limit only applies if it seems likely that most of the remaining errors will not be useful or they may be overly noisy. If N is negative, there is no limit. The default limit is 200.
1.20.12 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.

This setting will override the MYPY_CACHE_DIR environment variable if it is set.

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).

--sqlite-cache
Use an SQLite database to store the cache.

--cache-fine-grained
Include fine-grained dependency information in the cache for the mypy daemon.

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

--skip-cache-mtime-checks
Skip cache internal consistency checks based on mtime.

1.20.13 Advanced options

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.

--raise-exceptions
Raise exception on fatal error.

--custom-typing-module 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 standard library 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.

Note that this doesn’t affect third-party library stubs.
**--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-calls` 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.

### 1.20.14 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.

**--cobertura-xml-report** DIR

Causes mypy to generate a Cobertura XML type checking coverage report.

To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra `mypy[reports]`.

**--html-report** / **--xslt-html-report** DIR

Causes mypy to generate an HTML type checking coverage report.

To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra `mypy[reports]`.

**--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.

**--lineprecision-report** DIR

Causes mypy to generate a flat text file report with per-module statistics of how many lines are typechecked etc.
--txt-report / --xslt-txt-report DIR
Causes mypy to generate a text file type checking coverage report.

To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra mypy[reports].

--xml-report DIR
Causes mypy to generate an XML type checking coverage report.

To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra mypy[reports].

1.20.15 Miscellaneous

--install-types
This flag causes mypy to install known missing stub packages for third-party libraries using pip. It will display the pip command that will be run, and expects a confirmation before installing anything. For security reasons, these stubs are limited to only a small subset of manually selected packages that have been verified by the typeshed team. These packages include only stub files and no executable code.

If you use this option without providing any files or modules to type check, mypy will install stub packages suggested during the previous mypy run. If there are files or modules to type check, mypy first type checks those, and proposes to install missing stubs at the end of the run, but only if any missing modules were detected.

Note: This is new in mypy 0.900. Previous mypy versions included a selection of third-party package stubs, instead of having them installed separately.

--non-interactive
When used together with --install-types, this causes mypy to install all suggested stub packages using pip without asking for confirmation, and then continues to perform type checking using the installed stubs, if some files or modules are provided to type check.

This is implemented as up to two mypy runs internally. The first run is used to find missing stub packages, and output is shown from this run only if no missing stub packages were found. If missing stub packages were found, they are installed and then another run is performed.

--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.

--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.)
1.21 The mypy configuration file

Mypy supports reading configuration settings from a file. By default it uses the file mypy.ini with a fallback to .mypy.ini, then pypackage.toml, then 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 configuration files in the above mentioned order.

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.

Some flags support user home directory and environment variable expansion. To refer to the user home directory, use ~ at the beginning of the path. To expand environment variables use $VARNAME or ${VARNAME}.

1.21.1 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.
- 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 dotted_module_name.* matches dotted_module_name and any submodules (so foo.bar.* would match all of foo.bar, foo.bar.baz, and foo.bar.baz.quux).

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 configuration is:

1. Inline configuration in the source file
2. Sections with concrete module names (foo.bar)
3. Sections with “unstructured” wildcard patterns (foo.*.baz), with sections later in the configuration file overriding sections earlier.
4. Sections with “well-structured” wildcard patterns (foo.bar.*), with more specific overriding more general.
5. Command line options.
6. 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.

1.21.2 Per-module and global options

Some of the 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.

Some other options, as specified in their description, may only be set in the global section ([mypy]).

1.21.3 Inverting option values

Options that take a boolean value may be inverted by adding no_ to their name or by (when applicable) swapping their prefix from disallow to allow (and vice versa).

1.21.4 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.

```ini
# Global options:
[mypy]
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 two global options in the [mypy] section. These two options will:

1. Report an error whenever a function returns a value that is inferred to have type Any.
2. 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.

1.21. The mypy configuration file
2. Selectively disable the “function is returning any” warnings within myfile.bar only. This overrides the global default we set earlier.

3. Suppress any error messages generated when your codebase tries importing the module somewheres. This is useful if somewheres is some 3rd party library missing type hints.

### 1.21.5 Import discovery

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

**mypy**

**mypy_path**

**Type** 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. Multiple paths are always separated with a : or , regardless of the platform. User home directory and environment variables will be expanded.

Relative paths are treated relative to the working directory of the mypy command, not the config file. Use the MYPY_CONFIG_FILE_DIR environment variable to refer to paths relative to the config file (e.g. mypy_path = $MYPY_CONFIG_FILE_DIR/src).

This option may only be set in the global section ([mypy]).

**Note:** On Windows, use UNC paths to avoid using : (e.g. \127.0.0.1\X\MyDir where X is the drive letter).

**files**

**Type** comma-separated list of strings

A comma-separated list of paths which should be checked by mypy if none are given on the command line. Supports recursive file globbing using glob, where * (e.g. *.py) matches files in the current directory and **/ (e.g. **/*.py) matches files in any directories below the current one. User home directory and environment variables will be expanded.

This option may only be set in the global section ([mypy]).

**exclude**

**Type** regular expression

A regular expression that matches file names, directory names and paths which mypy should ignore while recursively discovering files to check. Use forward slashes (/) as directory separators on all platforms.

```python
[mypy]
exclude = (?x)(
    ^one\.py$ # files named "one.py"
    | two\.pyi$ # or files ending with "two.pyi"
    | ^three\. # or files starting with "three."
)
```

Crafting a single regular expression that excludes multiple files while remaining human-readable can be a challenge. The above example demonstrates one approach. (?x) enables the VERBOSE flag for the subsequent regular expression, which ignores most whitespace and supports comments. The above is equivalent to: (^one\.py$|two\.pyi$|^three\.).

For more details, see --exclude.

This option may only be set in the global section ([mypy]).
**Note:** Note that the TOML equivalent differs slightly. It can be either a single string (including a multi-line string) – which is treated as a single regular expression – or an array of such strings. The following TOML examples are equivalent to the above INI example.

Array of strings:

```toml
[tool.mypy]
exclude = [
    "^one\\.py$", # TOML's double-quoted strings require escaping backslashes
    'two\.pyi$', # but TOML's single-quoted strings do not
    '^three\.',
]
```

A single, multi-line string:

```toml
[tool.mypy]
exclude = '''(?x)(
    ^one\.py$ # files named "one.py"
    | two\.pyi$ # or files ending with "two.pyi"
    | ^three\. # or files starting with "three."
)''' # TOML's single-quoted strings do not require escaping backslashes
```

See [Using a pyproject.toml file](#).

**namespace_packages**

- **Type:** boolean
- **Default:** False

Enables PEP 420 style namespace packages. See the corresponding flag `--namespace-packages` for more information.

This option may only be set in the global section ([mypy]).

**explicit_package_bases**

- **Type:** boolean
- **Default:** False

This flag tells mypy that top-level packages will be based in either the current directory, or a member of the `MYPYPATH` environment variable or `mypy_path` config option. This option is only useful in conjunction with `namespace_packages`. See [Mapping file paths to modules](#) for details.

This option may only be set in the global section ([mypy]).

**ignore_missing_imports**

- **Type:** boolean
- **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

Type  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.

Using this option in a per-module section (potentially with a wildcard, as described at the top of this page) is a good way to prevent mypy from checking portions of your code.

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

Type  boolean
Default  False

Determines whether to respect the follow_imports setting even for stub (.pyi) files.

Used in conjunction with follow_imports=skip, this can be used to suppress the import of a module from typeshed, replacing it with Any.

Used in conjunction with follow_imports=error, this can be used to make any use of a particular typeshed module an error.

Note: This is not supported by the mypy daemon.

python_executable

Type  string

Specifies the path to the Python executable to inspect to collect a list of available PEP561 packages. User home directory and environment variables will be expanded. Defaults to the executable used to run mypy.

This option may only be set in the global section ([mypy]).

no_site_packages

Type  bool
Default  False

Disables using type information in installed packages (see PEP561). This will also disable searching for a usable Python executable. This acts the same as --no-site-packages command line flag.

no_silence_site_packages

Type  boolean
Default  False

Enables reporting error messages generated within installed packages (see PEP561 for more details on distributing type information). Those error messages are suppressed by default, since you are usually not able to control errors in 3rd party code.

This option may only be set in the global section ([mypy]).
1.21.6 Platform configuration

python_version

Type  string

Specifies the Python version used to parse and check the target program. The string should be in the format MAJOR.MINOR – for example 2.7. The default is the version of the Python interpreter used to run mypy.

This option may only be set in the global section ([mypy]).

platform

Type  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.

This option may only be set in the global section ([mypy]).

always_true

Type  comma-separated list of strings

Specifies a list of variables that mypy will treat as compile-time constants that are always true.

always_false

Type  comma-separated list of strings

Specifies a list of variables that mypy will treat as compile-time constants that are always false.

1.21.7 Disallow dynamic typing

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

disallow_any_unimported

Type  boolean

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

Type  boolean

Default  False

Disallows all expressions in the module that have type Any.

disallow_any_decorated

Type  boolean

Default  False

Disallows functions that have Any in their signature after decorator transformation.
disallow_any_explicit

Type  boolean
Default False

Disallow explicit Any in type positions such as type annotations and generic type parameters.

disallow_any.generics

Type  boolean
Default False

Disallow usage of generic types that do not specify explicit type parameters.

disallow_subclassing_any

Type  boolean
Default False

Disallow subclassing a value of type Any.

1.21.8 Untyped definitions and calls

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

disallow_untyped_calls

Type  boolean
Default False

Disallow calling functions without type annotations from functions with type annotations.

disallow_untyped_defs

Type  boolean
Default False

Disallow defining functions without type annotations or with incomplete type annotations.

disallow_incomplete_defs

Type  boolean
Default False

Disallow defining functions with incomplete type annotations.

check_untyped_defs

Type  boolean
Default False

Type-checks the interior of functions without type annotations.

disallow_untyped_decorators

Type  boolean
Default False

Reports an error whenever a function with type annotations is decorated with a decorator without annotations.
1.21.9 None and Optional handling

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

**no_implicit_optional**

- **Type**: boolean
- **Default**: False

Changes the treatment of arguments with a default value of None by not implicitly making their type Optional.

**strict_optional**

- **Type**: boolean
- **Default**: True

Enables or disables strict Optional checks. If False, mypy treats None as compatible with every type.

**Note**: This was False by default in mypy versions earlier than 0.600.

1.21.10 Configuring warnings

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

**warn_redundant_casts**

- **Type**: boolean
- **Default**: False

Warns about casting an expression to its inferred type.

This option may only be set in the global section ([mypy]).

**warn_unused_ignores**

- **Type**: boolean
- **Default**: False

Warns about unneeded # type: ignore comments.

**warn_no_return**

- **Type**: boolean
- **Default**: True

Shows errors for missing return statements on some execution paths.

**warn_return_any**

- **Type**: boolean
- **Default**: False

Shows a warning when returning a value with type Any from a function declared with a non- Any return type.

**warn_unreachable**

- **Type**: boolean
- **Default**: False
Shows a warning when encountering any code inferred to be unreachable or redundant after performing type analysis.

### 1.21.11 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**

- **Type**: boolean
- **Default**: True

Shows errors related to strict `None` checking, if the global `strict_optional` flag is enabled.

**ignore_errors**

- **Type**: boolean
- **Default**: False

Ignores all non-fatal errors.

### 1.21.12 Miscellaneous strictness flags

For more information, see the Miscellaneous strictness flags section of the command line docs.

**allow_untypedGlobals**

- **Type**: boolean
- **Default**: False

Causes mypy to suppress errors caused by not being able to fully infer the types of global and class variables.

**allow_redefinition**

- **Type**: boolean
- **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. Example where this can be useful:

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

The variable must be used before it can be redefined:

```python
def process(items: list[str]) -> None:
    items = "mypy"  # invalid redefinition to str because the variable hasn't been used yet
    print(items)
    items = "100"  # valid, items now has type str
    items = int(items)  # valid, items now has type int
```
local_partial_types

  Type  boolean
  Default  False

  Disallows inferring variable type for `None` from two assignments in different scopes. This is always implicitly enabled when using the `mypy daemon`.

disable_error_code

  Type  comma-separated list of strings

  Allows disabling one or multiple error codes globally.

enable_error_code

  Type  comma-separated list of strings

  Allows enabling one or multiple error codes globally.

  Note: This option will override disabled error codes from the disable_error_code option.

implicit_reexport

  Type  boolean
  Default  True

  By default, imported values to a module are treated as exported and mypy allows other modules to import them. When false, mypy will not re-export unless the item is imported using from-as or is included in `__all__`. Note that mypy treats stub files as if this is always disabled. For example:

```python
# This won't re-export the value
from foo import bar
# This will re-export it as bar and allow other modules to import it
from foo import bar as bar
# This will also re-export bar
from foo import bar
__all__ = ['bar']
```

strict_concatenate

  Type  boolean
  Default  False

  Make arguments prepended via `Concatenate` be truly positional-only.

strict_equality

  Type  boolean
  Default  False

  Prohibit equality checks, identity checks, and container checks between non-overlapping types.

strict

  Type  boolean
  Default  False
Enable 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 `strict` may change over time.

### 1.21.13 Configuring error messages

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

These options may only be set in the global section ([mypy]).

**show_error_context**

- **Type**: boolean
- **Default**: False

Prefixes each error with the relevant context.

**show_column_numbers**

- **Type**: boolean
- **Default**: False

Shows column numbers in error messages.

**show_error_codes**

- **Type**: boolean
- **Default**: False

Shows error codes in error messages. See *Error codes* for more information.

**pretty**

- **Type**: boolean
- **Default**: False

Use visually nicer output in error messages: use soft word wrap, show source code snippets, and show error location markers.

**color_output**

- **Type**: boolean
- **Default**: True

Shows error messages with color enabled.

**error_summary**

- **Type**: boolean
- **Default**: True

Shows a short summary line after error messages.

**show_absolute_path**

- **Type**: boolean
- **Default**: False

Show absolute paths to files.
1.21.14 Incremental mode

These options may only be set in the global section ([mypy]).

**incremental**

Type boolean

Default True

Enables **incremental mode**.

**cache_dir**

Type string

Default .mypy_cache

Specifies the location where mypy stores incremental cache info. User home directory and environment variables will be expanded. This setting will be overridden by the MYPY_CACHE_DIR environment variable.

Note that the cache is only read when incremental mode is enabled but is always written to, unless the value is set to /dev/null (UNIX) or nul (Windows).

**sqlite_cache**

Type boolean

Default False

Use an SQLite database to store the cache.

**cache_fine_grained**

Type boolean

Default False

Include fine-grained dependency information in the cache for the mypy daemon.

**skip_version_check**

Type boolean

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.)

**skip_cache_mtime_checks**

Type boolean

Default False

Skip cache internal consistency checks based on mtime.
1.21.15 Advanced options

These options may only be set in the global section ([mypy]).

**plugins**

  Type  comma-separated list of strings

  A comma-separated list of mypy plugins. See *Extending mypy using plugins*.

**pdb**

  Type  boolean

  Default  False

  Invokes pdb on fatal error.

**show_traceback**

  Type  boolean

  Default  False

  Shows traceback on fatal error.

**raise_exceptions**

  Type  boolean

  Default  False

  Raise exception on fatal error.

**custom_typing_module**

  Type  string

  Specifies a custom module to use as a substitute for the *typing* module.

**custom_typeshed_dir**

  Type  string

  Specifies an alternative directory to look for stubs instead of the default typeshed directory. User home directory and environment variables will be expanded.

**warn_incomplete_stub**

  Type  boolean

  Default  False

  Warns about missing type annotations in typeshed. This is only relevant in combination with *disallow_untyped_defs* or *disallow_incomplete_defs*. 
1.21.16 Report generation

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

any_exprs_report

    Type  string
    Causes mypy to generate a text file report documenting how many expressions of type Any are present within your codebase.

cobertura_xml_report

    Type  string
    Causes mypy to generate a Cobertura XML type checking coverage report.
    To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra mypy[reports].

html_report / xslt_html_report

    Type  string
    Causes mypy to generate an HTML type checking coverage report.
    To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra mypy[reports].

linecount_report

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

linecoverage_report

    Type  string
    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.

lineprecision_report

    Type  string
    Causes mypy to generate a flat text file report with per-module statistics of how many lines are typechecked etc.

txt_report / xslt_txt_report

    Type  string
    Causes mypy to generate a text file type checking coverage report.
    To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra mypy[reports].

xml_report

    Type  string
    Causes mypy to generate an XML type checking coverage report.
    To generate this report, you must either manually install the lxml library or specify mypy installation with the setuptools extra mypy[reports].
1.21.17 Miscellaneous

These options may only be set in the global section ([mypy]).

**junit_xml**

Type: string

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.

**scripts_are_modules**

Type: boolean  
Default: False

Makes script \( x \) become module \( x \) instead of \( \text{__main__} \). This is useful when checking multiple scripts in a single run.

**warn_unused_configs**

Type: boolean  
Default: False

Warns about per-module sections in the config file that do not match any files processed when invoking mypy. (This requires turning off incremental mode using \( \text{incremental} = \text{False} \).)

**verbosity**

Type: integer  
Default: 0

Controls how much debug output will be generated. Higher numbers are more verbose.

1.21.18 Using a pyproject.toml file

Instead of using a mypy.ini file, a pyproject.toml file (as specified by PEP 518) may be used instead. A few notes on doing so:

- The [mypy] section should have tool. prepended to its name:
  - I.e., [mypy] would become [tool.mypy]
- The module specific sections should be moved into [[tool.mypyoverrides]] sections:
  - For example, [mypy-packagename] would become:

```toml
[[tool.mypy.overrides]]
module = 'packagename'
...
```

- Multi-module specific sections can be moved into a single [[tool.mypy.overrides]] section with a module property set to an array of modules:
  - For example, [mypy-packagename,packagename2] would become:
The following care should be given to values in the `pyproject.toml` files as compared to ini files:

- Strings must be wrapped in double quotes, or single quotes if the string contains special characters
- Boolean values should be all lower case

Please see the TOML Documentation for more details and information on what is allowed in a toml file. See PEP 518 for more information on the layout and structure of the `pyproject.toml` file.

### 1.21.19 Example pyproject.toml

Here is an example of a `pyproject.toml` file. To use this config file, place it at the root of your repo (or append it to the end of an existing `pyproject.toml` file) and run mypy.

```toml
# mypy global options:

[tool.mypy]
python_version = "2.7"
warn_return_any = true
warn_unused_configs = true
exclude = [
    "^file1\.py$",  # TOML literal string (single-quotes, no escaping necessary)
    "^file2\.\py$",  # TOML basic string (double-quotes, backslash and other characters need escaping)
]

# mypy per-module options:

[[tool.mypyoverrides]]
module = "mycode.foo.*"
disable_untyped_def $ = true

[[tool.mypyoverrides]]
module = "mycode.bar"
warn_return_any = false

[[tool.mypyoverrides]]
module = [
    "somelibrary",
    "some_other_library"
]
ignore_missing_imports = true
```
1.22 Inline configuration

Mypy supports setting per-file configuration options inside files themselves using # mypy: comments. For example:

```markdown
# mypy: disallow-any-generics
```

Inline configuration comments take precedence over all other configuration mechanisms.

1.22.1 Configuration comment format

Flags correspond to config file flags but allow hyphens to be substituted for underscores.
Values are specified using =, but = True may be omitted:

```markdown
# mypy: disallow-any-generics
# mypy: always-true=FOO
```

Multiple flags can be separated by commas or placed on separate lines. To include a comma as part of an option’s value, place the value inside quotes:

```markdown
# mypy: disallow-untyped-defs, always-true="FOO,BAR"
```

Like in the configuration file, options that take a boolean value may be inverted by adding no- to their name or by (when applicable) swapping their prefix from disallow to allow (and vice versa):

```markdown
# mypy: allow-untyped-defs, no-strict-optional
```

1.23 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 command-line interface of mypy daemon 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.
1.23.1 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 -- prog.py pkg/*.py
```

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

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.)

**Note:** Mypy 0.780 added support for following imports in dmypy (enabled by default). This functionality is still experimental. You can use `--follow-imports=skip` or `--follow-imports=error` to fall back to the stable functionality. See `Following imports` for details on how these work.

1.23.2 Daemon client commands

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 recheck` checks the same set of files as the most recent `check` or `recheck` command. (You can also use the `--update` and `--remove` options to alter the set of files, and to define which files should be processed.)
- `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.

1.23.3 Additional daemon flags

`--status-file FILE`

Use `FILE` as the status file for storing daemon runtime state. This is normally a JSON file that contains information about daemon process and connection. The default path is `.dmypy.json` in the current working directory.

`--log-file FILE`

Direct daemon stdout/stderr to `FILE`. This is useful for debugging daemon crashes, since the server traceback is not always printed by the client. This is available for the `start`, `restart`, and `run` commands.
--timeout TIMEOUT
Automatically shut down server after TIMEOUT seconds of inactivity. This is available for the start, restart, and run commands.

--update FILE
Re-check FILE, or add it to the set of files being checked (and check it). This option may be repeated, and it’s only available for the recheck command. By default, mypy finds and checks all files changed since the previous run and files that depend on them. However, if you use this option (and/or --remove), mypy assumes that only the explicitly specified files have changed. This is only useful to speed up mypy if you type check a very large number of files, and use an external, fast file system watcher, such as watchman or watchdog, to determine which files got edited or deleted. Note: This option is never required and is only available for performance tuning.

--remove FILE
Remove FILE from the set of files being checked. This option may be repeated. This is only available for the recheck command. See --update above for when this may be useful. Note: This option is never required and is only available for performance tuning.

--fswatcher-dump-file FILE
Collect information about the current internal file state. This is only available for the status command. This will dump JSON to FILE in the format {path: [modification_time, size, content_hash]}. This is useful for debugging the built-in file system watcher. Note: This is an internal flag and the format may change.

--perf-stats-file FILE
Write performance profiling information to FILE. This is only available for the check, recheck, and run commands.

--export-types
Store all expression types in memory for future use. This is useful to speed up future calls to dmypy inspect (but uses more memory). Only valid for check, recheck, and run command.

1.23.4 Static inference of annotations

The mypy daemon supports (as an experimental feature) statically inferring draft function and method type annotations. Use dmypy suggest FUNCTION to generate a draft signature in the format (param_type_1, param_type_2, .. .) -> ret_type (types are included for all arguments, including keyword-only arguments, *args and **kwargs).

This is a low-level feature intended to be used by editor integrations, IDEs, and other tools (for example, the mypy plugin for PyCharm), to automatically add annotations to source files, or to propose function signatures.

In this example, the function format_id() has no annotation:

```python
def format_id(user):
    return f"User: {user}"
```

root = format_id(0)

dmypy suggest uses call sites, return statements, and other heuristics (such as looking for signatures in base classes) to infer that format_id() accepts an int argument and returns a str. Use dmypy suggest module.format_id to print the suggested signature for the function.

More generally, the target function may be specified in two ways:

- By its fully qualified name, i.e. [package.]module.[class.]function.
- By its location in a source file, i.e. /path/to/file.py:line. The path can be absolute or relative, and line can refer to any line number within the function body.
This command can also be used to find a more precise alternative for an existing, imprecise annotation with some Any types.

The following flags customize various aspects of the dmypy suggest command.

--json
Output the signature as JSON, so that PyAnnotate can read it and add the signature to the source file. Here is what the JSON looks like:

```json
[{
  "func_name": "example.format_id",
  "line": 1,
  "path": "/absolute/path/to/example.py",
  "samples": 0,
  "signature": {
    "arg_types": ["int"],
    "return_type": "str"
  }
}]
```

--no-errors
Only produce suggestions that cause no errors in the checked code. By default, mypy will try to find the most precise type, even if it causes some type errors.

--no-any
Only produce suggestions that don’t contain Any types. By default mypy proposes the most precise signature found, even if it contains Any types.

--flex-any FRACTION
Only allow some fraction of types in the suggested signature to be Any types. The fraction ranges from 0 (same as --no-any) to 1.

--callsites
Only find call sites for a given function instead of suggesting a type. This will produce a list with line numbers and types of actual arguments for each call: /path/to/file.py:line: (arg_type_1, arg_type_2, ...).

--use-fixme NAME
Use a dummy name instead of plain Any for types that cannot be inferred. This may be useful to emphasize to a user that a given type couldn’t be inferred and needs to be entered manually.

--max-guesses NUMBER
Set the maximum number of types to try for a function (default: 64).

1.23.5 Statically inspect expressions

The daemon allows to get declared or inferred type of an expression (or other information about an expression, such as known attributes or definition location) using dmypy inspect LOCATION command. The location of the expression should be specified in the format path/to/file.py:line:column[:end_line:end_column]. Both line and column are 1-based. Both start and end position are inclusive. These rules match how mypy prints the error location in error messages.

If a span is given (i.e. all 4 numbers), then only an exactly matching expression is inspected. If only a position is given (i.e. 2 numbers, line and column), mypy will inspect all expressions, that include this position, starting from the innermost one.

Consider this Python code snippet:

```python
def foo(x: int, longer_name: str) -> None:
    x
    longer_name
```
Here to find the type of `x` one needs to call `dmypy inspect src.py:2:5:2:5` or `dmypy inspect src.py:2:5`. While for `longer_name` one needs to call `dmypy inspect src.py:3:5:3:15` or, for example, `dmypy inspect src.py:3:10`. Please note that this command is only valid after daemon had a successful type check (without parse errors), so that types are populated, e.g. using `dmypy check`. In case where multiple expressions match the provided location, their types are returned separated by a newline.

Important note: it is recommended to check files with `--export-types` since otherwise most inspections will not work without `--force-reload`.

```bash
--show INSPECTION
```

What kind of inspection to run for expression(s) found. Currently the supported inspections are:

- `type` (default): Show the best known type of a given expression.
- `attrs`: Show which attributes are valid for an expression (e.g. for auto-completion). Format is `{"Base1": ["name_1", "name_2", ...]; "Base2": ...}`. Names are sorted by method resolution order. If expression refers to a module, then module attributes will be under key like `"<full.module.name>"`.
- `definition` (experimental): Show the definition location for a name expression or member expression. Format is `path/to/file.py:line:column:Symbol`. If multiple definitions are found (e.g. for a Union attribute), they are separated by comma.

```bash
--verbose
```

Increase verbosity of types string representation (can be repeated). For example, this will print fully qualified names of instance types (like `"builtins.str"`), instead of just a short name (like `"str"`).

```bash
--limit NUM
```

If the location is given as `line:column`, this will cause daemon to return only at most `NUM` inspections of innermost expressions. Value of 0 means no limit (this is the default). For example, if one calls `dmypy inspect src.py:4:10 --limit=1` with this code

```python
def foo(x: int) -> str: ..
def bar(x: str) -> None: ...
baz: int
bar(foo(baz))
```

This will output just one type "int" (for `baz` name expression). While without the limit option, it would output all three types: "int", "str", and "None".

```bash
--include-span
```

With this option on, the daemon will prepend each inspection result with the full span of corresponding expression, formatted as `1:2:1:4 -> "int"`. This may be useful in case multiple expressions match a location.

```bash
--include-kind
```

With this option on, the daemon will prepend each inspection result with the kind of corresponding expression, formatted as `NameExpr -> "int"`. If both this option and `--include-span` are on, the kind will appear first, for example `NameExpr:1:2:1:4 -> "int"`.

```bash
--include-object-attrs
```

This will make the daemon include attributes of `object` (excluded by default) in case of an `atts` inspection.

```bash
--union-attrs
```

Include attributes valid for some of possible expression types (by default an intersection is returned). This is useful for union types of type variables with values. For example, with this code:

```python
from typing import Union
```
class A:
    x: int
    z: int
class B:
    y: int
    z: int
var: Union[A, B]

The command `dmypy inspect --show attrs src.py:10:1` will return 
"A": ["z"], "B": ["z"],
while with `--union-attrs` it will return 
"A": ["x", "z"], "B": ["y", "z"].

--force-reload

Force re-parsing and re-type-checking file before inspection. By default this is done only when needed (for example file was not loaded from cache or daemon was initially run without `--export-types` mypy option), since reloading may be slow (up to few seconds for very large files).

1.24 Using installed packages

Packages installed with pip can declare that they support type checking. For example, the `aiohttp` package has built-in support for type checking.

Packages can also provide stubs for a library. For example, `types-requests` is a stub-only package that provides stubs for the `requests` package. Stub packages are usually published from `typeshed`, a shared repository for Python library stubs, and have a name of form `types-<library>`. Note that many stub packages are not maintained by the original maintainers of the package.

The sections below explain how mypy can use these packages, and how you can create such packages.

Note: PEP 561 specifies how a package can declare that it supports type checking.

Note: New versions of stub packages often use type system features not supported by older, and even fairly recent mypy versions. If you pin to an older version of mypy (using `requirements.txt`, for example), it is recommended that you also pin the versions of all your stub package dependencies.

Note: Starting in mypy 0.900, most third-party package stubs must be installed explicitly. This decouples mypy and stub versioning, allowing stubs to updated without updating mypy. This also allows stubs not originally included with mypy to be installed. Earlier mypy versions included a fixed set of stubs for third-party packages.
1.24.1 Using installed packages with mypy (PEP 561)

Typically mypy will automatically find and use installed packages that support type checking or provide stubs. This requires that you install the packages in the Python environment that you use to run mypy. As many packages don’t support type checking yet, you may also have to install a separate stub package, usually named types-<library>. (See Missing imports for how to deal with libraries that don’t support type checking and are also missing stubs.)

If you have installed typed packages in another Python installation or environment, mypy won’t automatically find them. One option is to install another copy of those packages in the environment in which you use to run mypy. Alternatively, you can use the --python-executable flag to point to the target Python 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 don’t want to use installed packages that provide type information at all, use the --no-site-packages flag to disable searching for installed packages.

Note that stub-only packages 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 (see PEP 561: Stub-only Packages for more information).

1.24.2 Creating PEP 561 compatible packages

Note: You can generally ignore this section unless you maintain a package on PyPI, or want to publish type information for an existing PyPI package.

PEP 561 describes three main ways to distribute type information:

1. A package has inline type annotations in the Python implementation.
2. A package ships stub files with type information alongside the Python implementation.
3. A package ships type information for another package separately as stub files (also known as a “stub-only package”).

If you want to create a stub-only package for an existing library, the simplest way is to contribute stubs to the typeshed repository, and a stub package will automatically be uploaded to PyPI.

If you would like to publish a library package to a package repository yourself (e.g. on 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 file in their package directory. For example, here is a typical directory structure:

```python
setup.py
package_a/
    __init__.py
    lib.py
    py.typed
```

The setup.py file could look like this:

```python
from distutils.core import setup

setup(
    name="SuperPackageA",
    author="Me",
    version="0.1",
```

(continues on next page)
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:

```python
setup.py
package_b/
    __init__.py
    lib.py
    lib.pyi
    py.typed
```

The `setup.py` file might look like this:

```python
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 the `lib.pyi` stub file 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:

```python
setup.py
package_c-stubs/
    __init__.pyi
    lib.pyi
```

The `setup.py` might look like this:

```python
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"]
)
```
The instructions above are enough to ensure that the built wheels contain the appropriate files. However, to ensure inclusion inside the sdist (.tar.gz archive), you may also need to modify the inclusion rules in your `MANIFEST.in`:

```bash
global-include *.pyi
global-include *.typed
```

## 1.25 Extending and integrating mypy

### 1.25.1 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:

```python
import sys
from mypy import api

result = api.run(sys.argv[1:])

if result[0]:
    print('
Type checking report:
')
    print(result[0])  # stdout

if result[1]:
    print('
Error report:
')
    print(result[1])  # stderr

print('
Exit status:', result[2])
```

### 1.25.2 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 signal. In particular, there are no guarantees about backwards compatibility.
Backwards incompatible changes may be made without a deprecation period, but we will announce them in the plugin API changes announcement issue.

### 1.25.3 Configuring mypy to use plugins

Plugins are Python files that can be specified in a mypy config file using the `plugins` option and one of the two formats: relative or absolute path 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 the following sections we describe the basics of the plugin system with some examples. For more technical details, please read the docstrings in `mypy/plugin.py` in mypy source code. Also you can find good examples in the bundled plugins located in `mypy/plugins`.

### 1.25.4 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.plugin.Plugin`:

```python
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 the given full name. See the full plugin hook method 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.

### 1.25. Extending and integrating mypy

151
1.25.5 Current list of plugin hooks

```
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_function_signature_hook` is used to adjust the signature of a function.

`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_attribute_hook()` is similar to above, but for attributes on classes rather than instances. Unlike above, this does not have special casing for `__getattr__` or `__getattribute__`.

`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 dataclasses import dataclass

@dataclass  # built-in plugin adds `__init__` method here
class User:
```

(continues on next page)
name: str

user = User(name='example')  # 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:

```python
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.

get_additional_deps() can be used to add new dependencies for a module. It is called before semantic analysis. For example, this can be used if a library has dependencies that are dynamically loaded based on configuration information.

report_config_data() can be used if the plugin has some sort of per-module configuration that can affect typechecking. In that case, when the configuration for a module changes, we want to invalidate mypy’s cache for that module so that it can be rechecked. This hook should be used to report to mypy any relevant configuration data, so that mypy knows to recheck the module if the configuration changes. The hooks should return data encodable as JSON.

1.25.6 Notes about the semantic analyzer

Mypy 0.710 introduced a new semantic analyzer, and the old semantic analyzer was removed in mypy 0.730. Support for the new semantic analyzer required some changes to existing plugins. Here is a short summary of the most important changes:

- The order of processing AST nodes is different. Code outside functions is processed first, and functions and methods are processed afterwards.
- Each AST node can be processed multiple times to resolve forward references. The same plugin hook may be called multiple times, so they need to be idempotent.
- The anal_type() API method returns None if some part of the type is not available yet due to forward references, for example.
- When looking up symbols, you may encounter placeholder nodes that are used for names that haven’t been fully processed yet. You’ll generally want to request another semantic analysis iteration by deferring in that case.

See the docstring at the top of mypy/plugin.py for more details.
1.26 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:

```python
from other_module import dynamic

BORDER_WIDTH = 15

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

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

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

```python
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 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.
1.26.1 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.

Stubgen applies heuristics to avoid generating stubs for submodules that include tests or vendored third-party packages.

1.26.2 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 only 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 causes unwanted side effects, such as the running of tests. Stubgen tries to skip test modules even without this option, but this does not always work.

- `--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.

1.26.3 Additional flags

-h, --help
Show help message and exit.

--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).

--export-less
Don’t export all names imported from other modules within the same package. Instead, only export imported names that are not referenced in the module that contains the import.

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

-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.

-v, --verbose
Produce more verbose output.

-q, --quiet
Produce less verbose output.

1.27 Automatic stub testing (stubtest)

Stub files are files containing type annotations. See PEP 484 for more motivation and details.

A common problem with stub files is that they tend to diverge from the actual implementation. Mypy includes the stubtest tool that can automatically check for discrepancies between the stubs and the implementation at runtime.

1.27.1 What stubtest does and does not do

Stubtest will import your code and introspect your code objects at runtime, for example, by using the capabilities of the inspect module. Stubtest will then analyse the stub files, and compare the two, pointing out things that differ between stubs and the implementation at runtime.

It’s important to be aware of the limitations of this comparison. Stubtest will not make any attempt to statically analyse your actual code and relies only on dynamic runtime introspection (in particular, this approach means stubtest works well with extension modules). However, this means that stubtest has limited visibility; for instance, it cannot tell if a return type of a function is accurately typed in the stubs.

For clarity, here are some additional things stubtest can’t do:
• Type check your code – use `mypy` instead
• Generate stubs – use `stubgen` or `pyright --createstub` instead
• Generate stubs based on running your application or test suite – use `monkeytype` instead
• Apply stubs to code to produce inline types – use `retype` or `libcst` instead

In summary, `stubtest` works very well for ensuring basic consistency between stubs and implementation or to check for stub completeness. It’s used to test Python’s official collection of library stubs, `typeshed`.

### 1.27.2 Example

Here’s a quick example of what `stubtest` can do:

```bash
$ python3 -m pip install mypy

$ cat library.py
x = "hello, stubtest"

def foo(x=None):
    print(x)

$ cat library.pyi
x: int
def foo(x: int) -> None: ...

$ python3 -m mypy.stubtest library
error: library.foo is inconsistent, runtime argument "x" has a default value but stub doesn’t. Stub: at line 3
Runtime: at line 3 in file ~/library.py
def (x=None)
error: library.x variable differs from runtime type Literal['hello, stubtest']

Stub: at line 1
builtins.int
Runtime: 'hello, stubtest'
```

### 1.27.3 Usage

Running `stubtest` can be as simple as `stubtest module_to_check`. Run `stubtest --help` for a quick summary of options.

Subtest must be able to import the code to be checked, so make sure that `mypy` is installed in the same environment as the library to be tested. In some cases, setting `PYTHONPATH` can help `stubtest` find the code to import.

Similarly, `stubtest` must be able to find the stubs to be checked. `stubtest` respects the `MYPYPATH` environment variable – consider using this if you receive a complaint along the lines of “failed to find stubs”.

Note that `stubtest` requires `mypy` to be able to analyse stubs. If `mypy` is unable to analyse stubs, you may get an error on the lines of “not checking stubs due to mypy build errors”. In this case, you will need to mitigate those errors before
stubtest will run. Despite potential overlap in errors here, stubtest is not intended as a substitute for running mypy directly.

If you wish to ignore some of stubtest’s complaints, stubtest supports a pretty handy allowlist system.

The rest of this section documents the command line interface of stubtest.

--concise
   Makes stubtest’s output more concise, one line per error

--ignore-missing-stub
   Ignore errors for stub missing things that are present at runtime

--ignore-positional-only
   Ignore errors for whether an argument should or shouldn’t be positional-only

--allowlist FILE
   Use file as an allowlist. Can be passed multiple times to combine multiple allowlists. Allowlists can be created with –generate-allowlist. Allowlists support regular expressions.

--generate-allowlist
   Print an allowlist (to stdout) to be used with –allowlist

--ignore-unused-allowlist
   Ignore unused allowlist entries

--mypy-config-file FILE
   Use specified mypy config file to determine mypy plugins and mypy path

--custom-typeshed-dir DIR
   Use the custom typeshed in DIR

--check-typeshed
   Check all stdlib modules in typeshed

--help
   Show a help message :-)
1.28.2 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:

```python
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:

```python
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:

```python
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).

__init__ method has no annotated arguments or return type annotation. __init__ is considered fully-annotated if at least one argument is annotated, while mypy will infer the return type as `None`. The implication is that, for a __init__ method that has no argument, you’ll have to explicitly annotate the return type as `None` to type-check this __init__ method:

```python
def foo(s: str) -> str:
    return s
```

class A():
    def __init__(self, value: str):
        # Return type inferred as None, considered as typed_
        self.value = value
        foo(1) # error: Argument 1 to "foo" has incompatible type "int"; expected "str"

class B():
    def __init__(self):
        # No argument is annotated, considered as untyped method
        foo(1) # No error!

class C():
```

(continues on next page)
def __init__(self) -> None:  # Must specify return type to type-check
    foo(1)  # error: Argument 1 to “foo” has incompatible type "int"; expected "str"

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.

```python
def foo() -> str:
    return None  # No error!
```

You may have disabled strict optional checking (see Disabling strict optional checking for more).

### 1.28.3 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:

```python
import frobnicate  # Error: No module "frobnicate"
frobnicate.start()
```

You can add a # type: ignore comment to tell mypy to ignore this error:

```python
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:** You can use the form # type: ignore[<code>] to only ignore specific errors on the line. This way you are less likely to silence unexpected errors that are not safe to ignore, and this will also document what the purpose of the comment is. See Error codes for more information.

**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__:

```python
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.

### 1.28.4 Ignoring a whole file

A `# type: ignore` comment at the top of a module (before any statements, including imports or docstrings) has the effect of ignoring the entire module.

```python
# type: ignore
import foo
foo.bar()
```

### 1.28.5 Issues with code at runtime

Idiomatic use of type annotations can sometimes run up against what a given version of Python considers legal code. These can result in some of the following errors when trying to run your code:

- `ImportError` from circular imports
- `NameError`: name "X" is not defined from forward references
- `TypeError`: 'type' object is not subscriptable from types that are not generic at runtime
- `ImportError` or `ModuleNotFoundError` from use of stub definitions not available at runtime
- `TypeError`: unsupported operand type(s) for |: 'type' and 'type' from use of new syntax

For dealing with these, see [Annotation issues at runtime](#).

### 1.28.6 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.

### 1.28.7 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:

```python
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):

```python
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!

1.28.8 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.

```python
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:

```python
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!
```

1.28.9 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:

```python
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:

  ```python
  new_lst: List[A] = [B(), B()]
lst = new_lst  # OK
  ```

- Make a copy of the right hand side:

  ```python
  lst = list(new_lst)  # Also OK
  ```

- Use immutable collections as annotations whenever possible:
```

```python
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
```

### 1.28.10 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
```

### 1.28.11 Complex type tests

Mypy can usually infer the types correctly when using `isinstance`, `issubclass`, or `type(obj) is some_class` type tests, and even *user-defined type guards*, but for other kinds of checks you may need to add an explicit type cast:

```
from typing import Sequence, cast

def find_first_str(a: Sequence[object]) -> str:
    index = next((i for i, s in enumerate(a) if isinstance(s, str)), -1)
    if index < 0:
        raise ValueError('No str found')

    found = a[index]  # Has type "object", despite the fact that we know it is "str"
    return cast(str, found)  # We need an explicit cast to make mypy happy
```

Alternatively, you can use an `assert` statement together with some of the supported type inference techniques:

```
from typing import Sequence

def find_first_str(a: Sequence[object]) -> str:
    index = next((i for i, s in enumerate(a) if isinstance(s, str)), -1)
    if index < 0:
        raise ValueError('No str found')

    found = a[index]  # Has type "object", despite the fact that we know it is "str"
    assert isinstance(found, str)  # Now, "found" will be narrowed to "str"
    return found  # No need for the explicit "cast()" anymore
```
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`.

Note: You can read more about type narrowing techniques *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.

### 1.28.12 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:

```python
import sys

# Distinguishing between different versions of Python:
if sys.version_info >= (3, 8):
    # Python 3.8+ specific definitions and imports
else:
    # Other 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:

```python
import sys

assert sys.platform != 'win32'

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

Some other expressions exhibit similar behavior; in particular, `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 type-checks if were run using Python 3.8, pass in `--python-version 3.8` from the command line. Note that you do not need to have Python 3.8 installed to perform this check.

To target a different operating system, use the `--platform PLATFORM` flag. For example, to verify your code type-checks if it were run in Windows, pass in `--platform win32`. See the documentation for `sys.platform` for examples of valid platform parameters.

### 1.28.13 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:

```python
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.

### 1.28.14 Silencing linters

In some cases, linters will complain about unused imports or code. In these cases, you can silence them with a comment after type comments, or on the same line as the import:

```
# to silence complaints about unused imports
from typing import List  # noqa
a = None  # type: List[int]
```

To silence the linter on the same line as a type comment put the linter comment after the type comment:

```
a = some_complex_thing()  # type: ignore  # noqa
```
1.28.15 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
```

1.28.16 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 workaround is to use an alias:

```
bytes_ = bytes
class Message:
    def bytes(self):
        ...
```

(continues on next page)
def register(self, path: bytes_):
    ...

### 1.28.17 Using a development mypy build

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

### 1.28.18 Variables vs type aliases

Mypy has both type aliases and variables with types like `Type[...]`. These are subtly different, and it’s important to understand how they differ to avoid pitfalls.

1. A variable with type `Type[...]` is defined using an assignment with an explicit type annotation:

   ```python
   class A: ...
   tp: Type[A] = A
   ```

2. You can define a type alias using an assignment without an explicit type annotation at the top level of a module:

   ```python
   class A: ...
   Alias = A
   ```

   You can also use `TypeAlias` (PEP 613) to define an explicit type alias:

   ```python
   from typing import TypeAlias
   class A: ...
   Alias: TypeAlias = A
   ```

   You should always use `TypeAlias` to define a type alias in a class body or inside a function.

   The main difference is that the target of an alias is precisely known statically, and this means that they can be used in type annotations and other type contexts. Type aliases can’t be defined conditionally (unless using supported Python version and platform checks):

   ```python
   class A: ...
   class B: ...

   if random() > 0.5:
       Alias = A
   else:
       # error: Cannot assign multiple types to name "Alias" without an
       # explicit "Type[...]" annotation
       Alias = B

   tp: Type[object]    # "tp" is a variable with a type object value
   ```
if random() > 0.5:
    tp = A
else:
    tp = B  # This is OK

def fun1(x: Alias) -> None: ...  # OK

def fun2(x: tp) -> None: ...  # Error: "tp" is not valid as a type

1.28.19 Incompatible overrides

It's unsafe to override a method with a more specific argument type, as it violates the Liskov substitution principle. For return types, it’s unsafe to override a method with a more general return type.

Other incompatible signature changes in method overrides, such as adding an extra required parameter, or removing an optional parameter, will also generate errors. The signature of a method in a subclass should accept all valid calls to the base class method. Mypy treats a subclass as a subtype of the base class. An instance of a subclass is valid everywhere where an instance of the base class is valid.

This example demonstrates both safe and unsafe overrides:

```python
from typing import Sequence, List, Iterable

class A:
    def test(self, t: Sequence[int]) -> Sequence[str]:
        ...  

class GeneralizedArgument(A):
    # A more general argument type is okay
    def test(self, t: Iterable[int]) -> Sequence[str]:  # OK
        ...

class NarrowerArgument(A):
    # A more specific argument type isn't accepted
    def test(self, t: List[int]) -> Sequence[str]:  # Error
        ...

class NarrowerReturn(A):
    # A more specific return type is fine
    def test(self, t: Sequence[int]) -> List[str]:  # OK
        ...

class GeneralizedReturn(A):
    # A more general return type is an error
    def test(self, t: Sequence[int]) -> Iterable[str]:  # Error
        ...

You can use # type: ignore[override] to silence the error. Add it to the line that generates the error, if you decide that type safety is not necessary:

class NarrowerArgument(A):
    def test(self, t: List[int]) -> Sequence[str]:  # type: ignore[override]
        ...
```
1.28.20 Unreachable code

Mypy may consider some code as *unreachable*, even if it might not be immediately obvious why. It’s important to note that mypy will *not* type check such code. Consider this example:

```python
class Foo:
    bar: str = ''

def bar() -> None:
    foo: Foo = Foo()
    return
    x: int = 'abc'  # Unreachable -- no error
```

It’s easy to see that any statement after `return` is unreachable, and hence mypy will not complain about the mis-typed code below it. For a more subtle example, consider this code:

```python
class Foo:
    bar: str = ''

def bar() -> None:
    foo: Foo = Foo()
    assert foo.bar is None
    x: int = 'abc'  # Unreachable -- no error
```

Again, mypy will not report any errors. The type of `foo.bar` is `str`, and mypy reasons that it can never be `None`. Hence the `assert` statement will always fail and the statement below will never be executed. (Note that in Python, `None` is not an empty reference but an object of type `None`.)

In this example mypy will go on to check the last line and report an error, since mypy thinks that the condition could be either True or False:

```python
class Foo:
    bar: str = ''

def bar() -> None:
    foo: Foo = Foo()
    if not foo.bar:
        return
    x: int = 'abc'  # Reachable -- error
```

If you use the `--warn-unreachable` flag, mypy will generate an error about each unreachable code block.

1.28.21 Narrowing and inner functions

Because closures in Python are late-binding (https://docs.python-guide.org/writing/gotchas/#late-binding-closures), mypy will not narrow the type of a captured variable in an inner function. This is best understood via an example:

```python
def foo(x: Optional[int]) -> Callable[[], int]:
    if x is None:
        x = 5
    print(x + 1)  # mypy correctly deduces x must be an int here
def inner() -> int:
    return x + 1  # but (correctly) complains about this line
```

(continues on next page)
To get this code to type check, you could assign `y = x` after `x` has been narrowed, and use `y` in the inner function, or add an assert in the inner function.

### 1.29 Supported Python features

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

- Unsupported Python features

#### 1.29.1 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.

### 1.30 Error codes

Mypy can optionally display an error code such as `[attr-defined]` after each error message. Error codes serve two purposes:

1. It’s possible to silence specific error codes on a line using `# type: ignore[code]`. This way you won’t accidentally ignore other, potentially more serious errors.
2. The error code can be used to find documentation about the error. The next two topics (Error codes enabled by default and Error codes for optional checks) document the various error codes mypy can report.

Most error codes are shared between multiple related error messages. Error codes may change in future mypy releases.

#### 1.30.1 Displaying error codes

Error codes are not displayed by default. Use `--show-error-codes` or config `show_error_codes = True` to display error codes. Error codes are shown inside square brackets:

```bash
$ mypy --show-error-codes prog.py
prog.py:1: error: "str" has no attribute "trim" [attr-defined]
```

It’s also possible to require error codes for `type: ignore` comments. See `ignore-without-code` for more information.
1.30.2 Silencing errors based on error codes

You can use a special comment `# type: ignore[code, ...]` to only ignore errors with a specific error code (or codes) on a particular line. This can be used even if you have not configured mypy to show error codes. Currently it’s only possible to disable arbitrary error codes on individual lines using this comment.

You can also use `--disable-error-code` to disable specific error codes globally.

This example shows how to ignore an error about an imported name mypy thinks is undefined:

```
# 'foo' is defined in 'foolib', even though mypy can't see the # definition.
from foolib import foo  # type: ignore[attr-defined]
```

1.30.3 Enabling specific error codes

There are command-line flags and config file settings for enabling certain optional error codes, such as `--disallow-untyped-defs`, which enables the `no-untyped-def` error code.

You can use `--enable-error-code` to enable specific error codes that don’t have a dedicated command-line flag or config file setting.

1.30.4 Per-module enabling/disabling error codes

You can use configuration file sections to enable or disable specific error codes only in some modules. For example, this `mypy.ini` config will enable non-annotated empty containers in tests, while keeping other parts of code checked in strict mode:

```
[mypy]
strict = True

[mypy-tests.*]
allow_untyped_defs = True
allow_untyped_calls = True
disable_error_code = var-annotated, has-type
```

Note that per-module enabling/disabling acts as override over the global options. So that you don’t need to repeat the error code lists for each module if you have them in global config section. For example:

```
[mypy]
enable_error_code = truthy-bool, ignore-without-code, unused-awaitable

[mypy-extensions.*]
disable_error_code = unused-awaitable
```

The above config will allow unused awaitables in extension modules, but will still keep the other two error codes enabled. The overall logic is following:

• Command line and/or config main section set global error codes
• Individual config sections adjust them per glob/module
• Inline `# mypy: ...` comments can further adjust them for a specific module

So one can e.g. enable some code globally, disable it for all tests in the corresponding config section, and then re-enable it with an inline comment in some specific test.
1.31 Error codes enabled by default

This section documents various errors codes that mypy can generate with default options. See Error codes for general documentation about error codes. Error codes for optional checks documents additional error codes that you can enable.

1.31.1 Check that attribute exists [attr-defined]

Mypy checks that an attribute is defined in the target class or module when using the dot operator. This applies to both getting and setting an attribute. New attributes are defined by assignments in the class body, or assignments to self.x in methods. These assignments don’t generate attr-defined errors.

Example:

```python
class Resource:
    def __init__(self, name: str) -> None:
        self.name = name

r = Resource('x')
print(r.name)  # OK
print(r.id)    # Error: "Resource" has no attribute "id" [attr-defined]
r.id = 5      # Error: "Resource" has no attribute "id" [attr-defined]
```

This error code is also generated if an imported name is not defined in the module in a from ... import statement (as long as the target module can be found):

```python
# Error: Module "os" has no attribute "non_existent" [attr-defined]
from os import non_existent
```

A reference to a missing attribute is given the Any type. In the above example, the type of non_existent will be Any, which can be important if you silence the error.

1.31.2 Check that attribute exists in each union item [union-attr]

If you access the attribute of a value with a union type, mypy checks that the attribute is defined for every type in that union. Otherwise the operation can fail at runtime. This also applies to optional types.

Example:

```python
from typing import Union
class Cat:
    def sleep(self) -> None: ...
    def miaow(self) -> None: ...

class Dog:
    def sleep(self) -> None: ...
    def follow_me(self) -> None: ...

def func(animal: Union[Cat, Dog]) -> None:
    # OK: 'sleep' is defined for both Cat and Dog
    animal.sleep()
    # Error: Item "Cat" of "Union[Cat, Dog]" has no attribute "follow_me" [union-attr]
    animal.follow_me()
```
You can often work around these errors by using `assert isinstance(obj, ClassName)` or `assert obj is not None` to tell mypy that you know that the type is more specific than what mypy thinks.

### 1.31.3 Check that name is defined [name-defined]

Mypy expects that all references to names have a corresponding definition in an active scope, such as an assignment, function definition or an import. This can catch missing definitions, missing imports, and typos.

This example accidentally calls `sort()` instead of `sorted()`:

```python
x = sort([3, 2, 4])  # Error: Name "sort" is not defined [name-defined]
```

### 1.31.4 Check arguments in calls [call-arg]

Mypy expects that the number and names of arguments match the called function. Note that argument type checks have a separate error code `arg-type`.

Example:

```python
from typing import Sequence

def greet(name: str) -> None:
    print('hello', name)

greet('jack')  # OK
greet('jill', 'jack')  # Error: Too many arguments for "greet" [call-arg]
```

### 1.31.5 Check argument types [arg-type]

Mypy checks that argument types in a call match the declared argument types in the signature of the called function (if one exists).

Example:

```python
from typing import Optional

def first(x: list[int]) -> Optional[int]:
    return x[0] if x else 0

t = (5, 4)
# Error: Argument 1 to "first" has incompatible type "tuple[int, int]";
#   expected "list[int]" [arg-type]
print(first(t))
```
1.31.6 Check calls to overloaded functions [call-overload]

When you call an overloaded function, mypy checks that at least one of the signatures of the overload items match the argument types in the call.

Example:

```python
from typing import overload, Optional

@overload
def inc_maybe(x: None) -> None: ...

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

def inc_maybe(x: Optional[int]) -> Optional[int]:
    if x is None:
        return None
    else:
        return x + 1

inc_maybe(None)  # OK
inc_maybe(5)  # OK

# Error: No overload variant of "inc_maybe" matches argument type "float" [call-overload]
inc_maybe(1.2)
```

1.31.7 Check validity of types [valid-type]

Mypy checks that each type annotation and any expression that represents a type is a valid type. Examples of valid types include classes, union types, callable types, type aliases, and literal types. Examples of invalid types include bare integer literals, functions, variables, and modules.

This example incorrectly uses the function `log` as a type:

```python
def log(x: object) -> None:
    print('log:', repr(x))

# Error: Function "t.log" is not valid as a type [valid-type]
def log_all(objs: list[object], f: log) -> None:
    for x in objs:
        f(x)
```

You can use `Callable` as the type for callable objects:

```python
from typing import Callable

# OK
def log_all(objs: list[object], f: Callable[[object], None]) -> None:
    for x in objs:
        f(x)
```
1.31.8 Require annotation if variable type is unclear [var-annotated]

In some cases mypy can’t infer the type of a variable without an explicit annotation. Mypy treats this as an error. This typically happens when you initialize a variable with an empty collection or None. If mypy can’t infer the collection item type, mypy replaces any parts of the type it couldn’t infer with Any and generates an error.

Example with an error:

class Bundle:
    def __init__(self) -> None:
        # Error: Need type annotation for "items"
        # (hint: "items: list[type] = ...") [var-annotated]
        self.items = []

reveal_type(Bundle().items)  # list[Any]

To address this, we add an explicit annotation:

class Bundle:
    def __init__(self) -> None:
        self.items: list[str] = []  # OK

reveal_type(Bundle().items)  # list[str]

1.31.9 Check validity of overrides [override]

Mypy checks that an overridden method or attribute is compatible with the base class. A method in a subclass must accept all arguments that the base class method accepts, and the return type must conform to the return type in the base class (Liskov substitution principle).

Argument types can be more general is a subclass (i.e., they can vary contravariantly). The return type can be narrowed in a subclass (i.e., it can vary covariantly). It’s okay to define additional arguments in a subclass method, as long all extra arguments have default values or can be left out (*args, for example).

Example:

from typing import Optional, Union

class Base:
    def method(self, arg: int) -> Optional[int]:
        ...

class Derived(Base):
    def method(self, arg: Union[int, str]) -> int:  # OK
        ...

class DerivedBad(Base):
    # Error: Argument 1 of "method" is incompatible with "Base" [override]
    def method(self, arg: bool) -> int:
        ...

1.31. Error codes enabled by default 175
1.31.10 Check that function returns a value [return]

If a function has a non-None return type, mypy expects that the function always explicitly returns a value (or raises an exception). The function should not fall off the end of the function, since this is often a bug.

Example:

```python
# Error: Missing return statement [return]
def show(x: int) -> int:
    print(x)

# Error: Missing return statement [return]
def pred1(x: int) -> int:
    if x > 0:
        return x - 1

# OK
def pred2(x: int) -> int:
    if x > 0:
        return x - 1
    else:
        raise ValueError('not defined for zero')
```

1.31.11 Check that return value is compatible [return-value]

Mypy checks that the returned value is compatible with the type signature of the function.

Example:

```python
def func(x: int) -> str:
    # Error: Incompatible return value type (got "int", expected "str") [return-value]
    return x + 1
```

1.31.12 Check types in assignment statement [assignment]

Mypy checks that the assigned expression is compatible with the assignment target (or targets).

Example:

```python
class Resource:
    def __init__(self, name: str) -> None:
        self.name = name

r = Resource('A')

r.name = 'B'  # OK

# Error: Incompatible types in assignment (expression has type "int",
# variable has type "str") [assignment]
r.name = 5
```
1.31.13 Check type variable values [type-var]

Mypy checks that value of a type variable is compatible with a value restriction or the upper bound type.

Example:

```python
from typing import TypeVar
T1 = TypeVar('T1', int, float)

def add(x: T1, y: T1) -> T1:
    return x + y

add(4, 5.5)  # OK

# Error: Value of type variable "T1" of "add" cannot be "str" [type-var]
add('x', 'y')
```

1.31.14 Check uses of various operators [operator]

Mypy checks that operands support a binary or unary operation, such as + or ~. Indexing operations are so common that they have their own error code index (see below).

Example:

```python
# Error: Unsupported operand types for + ("int" and "str") [operator]
1 + 'x'
```

1.31.15 Check indexing operations [index]

Mypy checks that the indexed value in indexing operation such as x[y] supports indexing, and that the index expression has a valid type.

Example:

```python
a = {'x': 1, 'y': 2}

a['x']  # OK

# Error: Invalid index type "int" for "dict[str, int]"; expected type "str" [index]
print(a[1])

# Error: Invalid index type "bytes" for "dict[str, int]"; expected type "str" [index]
a[b'x'] = 4
```
1.31.16 Check list items [list-item]

When constructing a list using `[item, ...], mypy checks that each item is compatible with the list type that is inferred from the surrounding context.

Example:

```
# Error: List item 0 has incompatible type "int"; expected "str" [list-item]
a: list[str] = [0]
```

1.31.17 Check dict items [dict-item]

When constructing a dictionary using `{key: value, ...} or dict(key=value, ...), mypy checks that each key and value is compatible with the dictionary type that is inferred from the surrounding context.

Example:

```
# Error: Dict entry 0 has incompatible type "str": "str"; expected "str": "int" [dict-item]
d: dict[str, int] = {'key': 'value'}
```

1.31.18 Check TypedDict items [typeddict-item]

When constructing a TypedDict object, mypy checks that each key and value is compatible with the TypedDict type that is inferred from the surrounding context.

When getting a TypedDict item, mypy checks that the key exists. When assigning to a TypedDict, mypy checks that both the key and the value are valid.

Example:

```
from typing_extensions import TypedDict

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

# Error: Incompatible types (expression has type "float",
#       TypedDict item "x" has type "int") [typeddict-item]
p: Point = {'x': 1.2, 'y': 4}
```

1.31.19 Check that type of target is known [has-type]

Mypy sometimes generates an error when it hasn’t inferred any type for a variable being referenced. This can happen for references to variables that are initialized later in the source file, and for references across modules that form an import cycle. When this happens, the reference gets an implicit Any type.

In this example the definitions of x and y are circular:

```
class Problem:
    def set_x(self) -> None:
        # Error: Cannot determine type of "y" [has-type]
```
To work around this error, you can add an explicit type annotation to the target variable or attribute. Sometimes you can also reorganize the code so that the definition of the variable is placed earlier than the reference to the variable in a source file. Untangling cyclic imports may also help.

We add an explicit annotation to the \texttt{y} attribute to work around the issue:

```python
class Problem:
    def set_x(self) -> None:
        self.x = self.y  # OK

    def set_y(self) -> None:
        self.y: int = self.x  # Added annotation here
```

### 1.31.20 Check that import target can be found [import]

Mypy generates an error if it can’t find the source code or a stub file for an imported module.

Example:

```python
import acme
# Error: Cannot find implementation or library stub for module named 'acme' [import]
```

See \textit{Missing imports} for how to work around these errors.

### 1.31.21 Check that each name is defined once [no-redef]

Mypy may generate an error if you have multiple definitions for a name in the same namespace. The reason is that this is often an error, as the second definition may overwrite the first one. Also, mypy often can’t be able to determine whether references point to the first or the second definition, which would compromise type checking.

If you silence this error, all references to the defined name refer to the first definition.

Example:

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

class A:  # Error: Name "A" already defined on line 1 [no-redef]
    def __init__(self, x: str) -> None: ...

# Error: Argument 1 to "A" has incompatible type "str"; expected "int"
# (the first definition wins!)
A('x')
```
1.31.22 Check that called function returns a value [func-returns-value]

Mypy reports an error if you call a function with a `None` return type and don’t ignore the return value, as this is usually (but not always) a programming error.

In this example, the `if f()` check is always false since `f` returns `None`:

```python
def f() -> None:
    ...

# OK: we don't do anything with the return value
f()

# Error: "f" does not return a value [func-returns-value]
if f():
    print("not false")
```

1.31.23 Check instantiation of abstract classes [abstract]

Mypy generates an error if you try to instantiate an abstract base class (ABC). An abstract base class is a class with at least one abstract method or attribute. (See also `abc` module documentation)

Sometimes a class is made accidentally abstract, often due to an unimplemented abstract method. In a case like this you need to provide an implementation for the method to make the class concrete (non-abstract).

Example:

```python
from abc import ABCMeta, abstractmethod

class Persistent(metaclass=ABCMeta):
    @abstractmethod
def save(self) -> None: ...

class Thing(Persistent):
    def __init__(self) -> None:
        ...

    # No "save" method

# Error: Cannot instantiate abstract class "Thing" with abstract attribute "save"

t = Thing()
```

1.31.24 Check the target of NewType [valid-newtype]

The target of a `NewType` definition must be a class type. It can’t be a union type, `Any`, or various other special types. You can also get this error if the target has been imported from a module whose source mypy cannot find, since any such definitions are treated by mypy as values with `Any` types. Example:

```python
from typing import NewType

# The source for "acme" is not available for mypy
```
from acme import Entity  # type: ignore

# Error: Argument 2 to NewType(...) must be subclassable (got "Any") [valid-newtype]
UserEntity = NewType('UserEntity', Entity)

To work around the issue, you can either give mypy access to the sources for acme or create a stub file for the module. See Missing imports for more information.

**1.31.25 Check the return type of __exit__ [exit-return]**

If mypy can determine that __exit__ always returns False, mypy checks that the return type is not bool. The boolean value of the return type affects which lines mypy thinks are reachable after a with statement, since any __exit__ method that can return True may swallow exceptions. An imprecise return type can result in mysterious errors reported near with statements.

To fix this, use either typing_extensions.Literal[False] or None as the return type. Returning None is equivalent to returning False in this context, since both are treated as false values.

Example:

```python
class MyContext:
    ...
    def __exit__(self, exc, value, tb) -> bool:  # Error
        print('exit')
        return False
```

This produces the following output from mypy:

```
example.py:3: error: "bool" is invalid as return type for "__exit__" that always returns...
example.py:3: note: Use "typing_extensions.Literal[False]" as the return type or change...
example.py:3: note: If return type of "__exit__" implies that it may return True, the...
```

You can use Literal[False] to fix the error:

```python
from typing_extensions import Literal
class MyContext:
    ...
    def __exit__(self, exc, value, tb) -> Literal[False]:  # OK
        print('exit')
        return False
```

You can also use None:

```python
class MyContext:
    ...
    def __exit__(self, exc, value, tb) -> None:  # Also OK
        print('exit')
```

1.31. Error codes enabled by default
1.31.26 Check that naming is consistent [name-match]

The definition of a named tuple or a TypedDict must be named consistently when using the call-based syntax. Example:

```python
from typing import NamedTuple

# Error: First argument to namedtuple() should be "Point2D", not "Point"
Point2D = NamedTuple("Point", ["x", int], ["y", int])
```

1.31.27 Check that overloaded functions have an implementation [no-overload-impl]

Overloaded functions outside of stub files must be followed by a non overloaded implementation.

```python
from typing import overload

@overload
def func(value: int) -> int:
    ...

@overload
def func(value: str) -> str:
    ...

# presence of required function below is checked
def func(value):
    pass  # actual implementation
```

1.31.28 Check that coroutine return value is used [unused-coroutine]

Mypy ensures that return values of async def functions are not ignored, as this is usually a programming error, as the coroutine won’t be executed at the call site.

```python
async def f() -> None:
    ...

async def g() -> None:
    f()  # Error: missing await
    await f()  # OK
```

You can work around this error by assigning the result to a temporary, otherwise unused variable:

```python
_ = f()  # No error
```
1.31.29 Check types in assert_type [assert-type]

The inferred type for an expression passed to assert_type must match the provided type.

```python
from typing_extensions import assert_type
assert_type([1], list[int])  # OK
assert_type([1], list[str])  # Error
```

1.31.30 Report syntax errors [syntax]

If the code being checked is not syntactically valid, mypy issues a syntax error. Most, but not all, syntax errors are blocking errors: they can’t be ignored with a # type: ignore comment.

1.31.31 Miscellaneous checks [misc]

Mypy performs numerous other, less commonly failing checks that don’t have specific error codes. These use the misc error code. Other than being used for multiple unrelated errors, the misc error code is not special. For example, you can ignore all errors in this category by using # type: ignore[misc] comment. Since these errors are not expected to be common, it’s unlikely that you’ll see two different errors with the misc code on a single line – though this can certainly happen once in a while.

Note: Future mypy versions will likely add new error codes for some errors that currently use the misc error code.

1.32 Error codes for optional checks

This section documents various errors codes that mypy generates only if you enable certain options. See Error codes for general documentation about error codes. Error codes enabled by default documents error codes that are enabled by default.

Note: The examples in this section use inline configuration to specify mypy options. You can also set the same options by using a configuration file or command-line options.

1.32.1 Check that type arguments exist [type-arg]

If you use --disallow-any-generics, mypy requires that each generic type has values for each type argument. For example, the types list or dict would be rejected. You should instead use types like list[int] or dict[str, int]. Any omitted generic type arguments get implicit Any values. The type list is equivalent to list[any], and so on.

Example:

```python
# mypy: disallow-any-generics

# Error: Missing type parameters for generic type "list" [type-arg]
```

(continues on next page)
def remove_dups(items: list) -> list:
...

1.32.2 Check that every function has an annotation [no-untyped-def]

If you use `--disallow-untyped-defs`, mypy requires that all functions have annotations (either a Python 3 annotation or a type comment).

Example:

```python
# mypy: disallow-untyped-defs

def inc(x):
    # Error: Function is missing a type annotation [no-untyped-def]
    return x + 1

def inc_ok(x: int) -> int:
    # OK
    return x + 1

class Counter:
    # Error: Function is missing a type annotation [no-untyped-def]
    def __init__(self):
        self.value = 0

class CounterOk:
    # OK: An explicit "-> None" is needed if "__init__" takes no arguments
    def __init__(self) -> None:
        self.value = 0
```

1.32.3 Check that cast is not redundant [redundant-cast]

If you use `--warn-redundant-casts`, mypy will generate an error if the source type of a cast is the same as the target type.

Example:

```python
# mypy: warn-redundant-casts
rom typing import cast

Count = int

def example(x: Count) -> int:
    # Error: Redundant cast to "int" [redundant-cast]
    return cast(int, x)
```
1.32.4 Check that comparisons are overlapping [comparison-overlap]

If you use `--strict-equality`, mypy will generate an error if it thinks that a comparison operation is always true or false. These are often bugs. Sometimes mypy is too picky and the comparison can actually be useful. Instead of disabling strict equality checking everywhere, you can use `# type: ignore[comparison-overlap]` to ignore the issue on a particular line only.

Example:

```python
# mypy: strict-equality

def is_magic(x: bytes) -> bool:
    # Error: Non-overlapping equality check (left operand type: "bytes",
    #    right operand type: "str") [comparison-overlap]
    return x == 'magic'
```

We can fix the error by changing the string literal to a bytes literal:

```python
# mypy: strict-equality

def is_magic(x: bytes) -> bool:
    return x == b'magic'  # OK
```

1.32.5 Check that no untyped functions are called [no-untyped-call]

If you use `--disallow-untyped-calls`, mypy generates an error when you call an unannotated function in an annotated function.

Example:

```python
# mypy: disallow-untyped-calls

def do_it() -> None:
    # Error: Call to untyped function "bad" in typed context [no-untyped-call]
    bad()

def bad():
    ...
```

1.32.6 Check that function does not return Any value [no-any-return]

If you use `--warn-return-any`, mypy generates an error if you return a value with an Any type in a function that is annotated to return a non-Any value.

Example:

```python
# mypy: warn-return-any

def fields(s):
    return s.split(',')

def first_field(x: str) -> str:
```

(continues on next page)
1.32.7 Check that types have no Any components due to missing imports [no-any-unimported]

If you use --disallow-any-unimported, mypy generates an error if a component of a type becomes Any because mypy couldn’t resolve an import. These “stealth” Any types can be surprising and accidentally cause imprecise type checking.

In this example, we assume that mypy can’t find the module animals, which means that Cat falls back to Any in a type annotation:

```python
# mypy: disallow-any-unimported
from animals import Cat  # type: ignore

# Error: Argument 1 to "feed" becomes "Any" due to an unfollowed import [no-any-unimported]
def feed(cat: Cat) -> None:
    ...
```

1.32.8 Check that statement or expression is unreachable [unreachable]

If you use --warn-unreachable, mypy generates an error if it thinks that a statement or expression will never be executed. In most cases, this is due to incorrect control flow or conditional checks that are accidentally always true or false.

```python
# mypy: warn-unreachable
def example(x: int) -> None:
    # Error: Right operand of "or" is never evaluated [unreachable]
    assert isinstance(x, int) or x == 'unused'
    return
    # Error: Statement is unreachable [unreachable]
    print('unreachable')
```

1.32.9 Check that expression is redundant [redundant-expr]

If you use --enable-error-code redundant-expr, mypy generates an error if it thinks that an expression is redundant.

```python
# Use "mypy --enable-error-code redundant-expr ...

def example(x: int) -> None:
    # Error: Left operand of "and" is always true [redundant-expr]
    if isinstance(x, int) and x > 0:
        pass
```

# Error: If condition is always true  [redundant-expr]
1 if isinstance(x, int) else 0

# Error: If condition in comprehension is always true  [redundant-expr]
[i for i in range(x) if isinstance(i, int)]

## 1.32.10 Check that expression is not implicitly true in boolean context [truthy-bool]

Warn when an expression whose type does not implement `__bool__` or `__len__` is used in boolean context, since unless implemented by a sub-type, the expression will always evaluate to true.

```python
# Use "mypy --enable-error-code truthy-bool ..."

class Foo:
    pass
foo = Foo()
# Error: "foo" has type "Foo" which does not implement __bool__ or __len__ so it could always be true in boolean context
if foo:
    ... 
```

This check might falsely imply an error. For example, `Iterable` does not implement `__len__` and so this code will be flagged:

```python
# Use "mypy --enable-error-code truthy-bool ..."

from typing import Iterable

def transform(items: Iterable[int]) -> Iterable[int]:
    # Error: "items" has type "Iterable[int]" which does not implement __bool__ or __len__ so it could always be true in boolean context [truthy-bool]
    if not items:
        return [42]
    return [x + 1 for x in items]
```

If called as `transform((int(s) for s in []))`, this function would not return `[42]` unlike what the author might have intended. Of course it’s possible that `transform` is only passed `list` objects, and so there is no error in practice. In such case, it might be prudent to annotate `items: Sequence[int]`.

This is similar in concept to ensuring that an expression’s type implements an expected interface (e.g. `Sized`), except that attempting to invoke an undefined method (e.g. `__len__`) results in an error, while attempting to evaluate an object in boolean context without a concrete implementation results in a truthy value.
1.32.11 Check that # type: ignore include an error code [ignore-without-code]

Warn when a # type: ignore comment does not specify any error codes. This clarifies the intent of the ignore and ensures that only the expected errors are silenced.

Example:

```python
# Use "mypy --enable-error-code ignore-without-code ..."

class Foo:
    def __init__(self, name: str) -> None:
        self.name = name

f = Foo('foo')

# This line has a typo that mypy can't help with as both:
# - the expected error 'assignment', and
# - the unexpected error 'attr-defined'
# are silenced.
# Error: "type: ignore" comment without error code (consider "type: ignore[attr-defined] ...") instead
f.nme = 42  # type: ignore

# This line warns correctly about the typo in the attribute name
# Error: "Foo" has no attribute "nme"; maybe "name"?
# Error: "type: ignore" comment without error code (consider "type: ignore[attr-defined] ...") instead
f.nme = 42  # type: ignore[assignment]
```

1.32.12 Check that awaitable return value is used [unused-awaitable]

If you use --enable-error-code unused-awaitable, mypy generates an error if you don’t use a returned value that defines __await__.

Example:

```python
# Use "mypy --enable-error-code unused-awaitable ..."

import asyncio

async def f() -> int: ...

async def g() -> None:
    # Error: Value of type "Task[int]" must be used
    # Are you missing an await?
    asyncio.create_task(f())

async def g() -> None:
    _ = asyncio.create_task(f())  # No error
```
1.33 Additional features

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

1.33.1 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:

```python
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:

```python
from dataclasses import dataclass

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

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

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:

```python
from dataclasses import dataclass
from typing import Generic, TypeVar

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
```
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:

```python
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

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
```

1.33.2 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:

```python
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

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.

1.33.3 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.

**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.

**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.

**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!
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.

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).

1.33.4 Extended Callable types

Note: This feature is deprecated. You can use `callback protocols` as a replacement.

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:

```python
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:

```python
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.

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

def VarArg(type=Any):
```

(continues on next page)
# A *args-style variadic positional argument.
# A single VarArg() specifier represents all remaining
# positional arguments.

```python
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

```python
MyFunc = Callable[[int, str, int], float]
```
is equivalent to the following:

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

A `Callable` with unspecified argument types, such as

```python
MyOtherFunc = Callable[... , int]
```
is (roughly) equivalent to

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

Note: Each of the functions above currently just returns its `type` argument at runtime, 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.

## 1.34 Frequently Asked Questions

### 1.34.1 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.

1.34.2 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.

1.34.3 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.

1.34.4 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.

1.34.5 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.
1.34.6 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.

1.34.7 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.

1.34.8 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.

1.34.9 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.

1.34.10 Does it run on PyPy?

Somewhat. With PyPy 3.8, mypy is at least able to type check itself. With older versions of PyPy, mypy relies on typed-ast, which uses several APIs that PyPy does not support (including some internal CPython APIs).

1.34.11 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, website 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
TWO

INDICES AND TABLES

• genindex
• search
Symbols

-v  mypy command line option, 114
--allow-redefinition  mypy command line option, 119
--allow-untyped-globals  mypy command line option, 119
--allowlist  stubtest command line option, 158
--always-false  mypy command line option, 116
--always-true  mypy command line option, 116
--any-exprs-report  mypy command line option, 124
--cache-dir  mypy command line option, 123
--cache-fine-grained  mypy command line option, 123
--callsites  dmypy command line option, 145
--check-typeshed  stubtest command line option, 158
--check-untyped-defs  mypy command line option, 117
--cobertura-xml-report  mypy command line option, 124
--command  mypy command line option, 114
--concise  stubtest command line option, 158
--config-file  mypy command line option, 114
--custom-typeshed-dir  mypy command line option, 123
--custom-typing-module  stubtest command line option, 158
--disable-error-code  mypy command line option, 114
--disallow-any-decorated  mypy command line option, 117
--disallow-any-explicit  mypy command line option, 117
--disallow-any-expr  mypy command line option, 117
--disallow-any-generics  mypy command line option, 117
--disallow-any-unimported  mypy command line option, 117
--disallow-incomplete-defs  mypy command line option, 117
--disallow-subclassing-any  mypy command line option, 117
--disallow-untyped-calls  mypy command line option, 117
--disallow-untyped-decorators  mypy command line option, 118
--disallow-untyped-defs  mypy command line option, 117
--doc-dir  stubgen command line option, 155
--enable-error-code  mypy command line option, 121
--exclude  mypy command line option, 114
--explicit-package-bases  mypy command line option, 115
--export-less  stubgen command line option, 156
--export-types  dmypy command line option, 144
--fast-module-lookup  mypy command line option, 116
--find-occurrences  mypy command line option, 125
--flex-any  dmypy command line option, 145
--follow-imports  mypy command line option, 115
--force-reload  dmypy command line option, 147
--fswatcher-dump-file  dmypy command line option, 144
--generate-allowlist
  stubtest command line option, 158
--help
  mypy command line option, 114
  stubgen command line option, 156
  stubtest command line option, 158
--html-report
  mypy command line option, 124
--ignore-errors
  stubgen command line option, 156
--ignore-missing-imports
  mypy command line option, 115
--ignore-missing-stub
  stubtest command line option, 158
--ignore-positional-only
  stubtest command line option, 158
--ignore-unused-allowlist
  stubtest command line option, 158
--include-kind
  dmypy command line option, 146
--include-object-attrs
  dmypy command line option, 146
--include-private
  stubgen command line option, 156
--include-span
  dmypy command line option, 146
--install-types
  mypy command line option, 125
--json
  dmypy command line option, 145
--junit-xml
  mypy command line option, 125
--limit
  dmypy command line option, 146
--linecount-report
  mypy command line option, 124
--linecoverage-report
  mypy command line option, 124
--lineprecision-report
  mypy command line option, 124
--local-partial-types
  mypy command line option, 120
--log-file
  dmypy command line option, 143
--max-guesses
  dmypy command line option, 145
--module
  mypy command line option, 113
  stubgen command line option, 155
--mypy-config-file
  stubtest command line option, 158
--namespace-packages
  mypy command line option, 115
--no-any
  dmypy command line option, 145
--no-color-output
  mypy command line option, 122
--no-error-summary
  mypy command line option, 122
--no-errors
  dmypy command line option, 145
--no-implicit-optional
  mypy command line option, 118
--no-implicit-reexport
  mypy command line option, 120
--no-import
  stubgen command line option, 155
--no-incremental
  mypy command line option, 123
--no-silence-site-packages
  mypy command line option, 116
--no-site-packages
  mypy command line option, 115
--no-strict-optional
  mypy command line option, 118
--no-warn-no-return
  mypy command line option, 118
--non-interactive
  mypy command line option, 125
--output
  stubgen command line option, 156
--package
  mypy command line option, 113
  stubgen command line option, 155
--parse-only
  stubgen command line option, 155
--pdb
  mypy command line option, 123
--perf-stats-file
  dmypy command line option, 144
--platform
  mypy command line option, 116
--pretty
  mypy command line option, 122
--python-executable
  mypy command line option, 115
--python-version
  mypy command line option, 116
--quiet
  stubgen command line option, 156
--raise-exceptions
  mypy command line option, 123
--remove
  dmypy command line option, 144
--scripts-are-modules
  mypy command line option, 125
--search-path
  stubgen command line option, 156
--shadow-file
  mypy command line option, 124
--show
  dmypy command line option, 146
--show-absolute-path
  mypy command line option, 122
--show-column-numbers
  mypy command line option, 122
--show-error-codes
  mypy command line option, 122
--show-error-context
  mypy command line option, 122
--show-error-end
  mypy command line option, 122
--show-traceback
  mypy command line option, 122
--skip-cache-mtime-checks
  mypy command line option, 123
--skip-version-check
  mypy command line option, 123
--soft-error-limit
  mypy command line option, 122
--sqlite-cache
  mypy command line option, 123
--status-file
  dmypy command line option, 143
--strict
  mypy command line option, 121
--strict-equality
  mypy command line option, 120
--tb
  mypy command line option, 123
--timeout
  mypy command line option, 123
--txt-report
  mypy command line option, 124
--union-attrs
  dmypy command line option, 146
--update
  dmypy command line option, 144
--use-fixme
  dmypy command line option, 145
--verbose
  dmypy command line option, 146
  mypy command line option, 114
  stubgen command line option, 156
--version
  mypy command line option, 114
--warn-incomplete-stub
  mypy command line option, 123
--warn-redundant-casts
  mypy command line option, 118
--warn-return-any
  mypy command line option, 119
--warn-unreachable
  mypy command line option, 119
--warn-unused-configs
  mypy command line option, 114
--warn-unused-ignores
  mypy command line option, 118
--xml-report
  mypy command line option, 125
-c
  mypy command line option, 114
-h
  mypy command line option, 114
  stubgen command line option, 156
-m
  mypy command line option, 113
  stubgen command line option, 155
-o
  stubgen command line option, 156
-p
  mypy command line option, 113
  stubgen command line option, 155
-q
  stubgen command line option, 156
-v
  mypy command line option, 114
  stubgen command line option, 156

A
allow_redefinition
  configuration value, 134
allow_untyped_globals
  configuration value, 134
always_false
  configuration value, 131
always_true
  configuration value, 131
any_exprs_report
  configuration value, 139

C
cache_dir
  configuration value, 137
cache_fine_grained
  configuration value, 137
check_untyped_defs
  configuration value, 132
cobertura_xml_report
  configuration value, 139
color_output
  configuration value, 136
configuration value
  allow_redefinition, 134
  allow_untyped_globals, 134
  always_false, 131
always_true, 131
any_exprs_report, 139
cache_dir, 137
cache_fine_grained, 137
check_untyped_defs, 132
cobertura_xml_report, 139
color_output, 136
custom_typeshed_dir, 138
custom_typing_module, 138
disable_error_code, 135
disallow_any_decorated, 131
disallow_any_explicit, 131
disallow_any_expr, 131
disallow_any_generics, 132
disallow_any_unimported, 131
disallow_incomplete_defs, 132
disallow_subclassing_any, 132
disallow_untyped_calls, 132
disallow_untyped_decorators, 132
disallow_untyped_defs, 132
enable_error_code, 135
error_summary, 136
exclude, 128
explicit_package_bases, 129
files, 128
follow_imports, 129
follow_imports_for_stubs, 130
html_report / xslt_html_report, 139
ignore_errors, 134
ignore_missing_imports, 129
implicit_reexport, 135
incremental, 137
junit_xml, 140
linecount_report, 139
linecoverage_report, 139
lineprecision_report, 139
local_partial_types, 134
mypy_path, 128
namespace_packages, 129
no_implicit_optional, 133
no_silence_site_packages, 130
no_site_packages, 130
pdb, 138
platform, 131
plugins, 138
pretty, 136
python_executable, 130
python_version, 131
raise_exceptions, 138
scripts_are_modules, 140
show_absolute_path, 136
show_column_numbers, 136
show_error_codes, 136
show_error_context, 136
show_none_errors, 134
show_traceback, 138
skip_cache_mtime_checks, 137
skip_version_check, 137
sqlite_cache, 137
strict, 135
strict_concatenate, 135
strict_equality, 135
strict_optional, 133
txt_report / xslt_txt_report, 139
verbosity, 140
warn_incomplete_stub, 138
warn_no_return, 133
warn_redundant_casts, 133
warn_return_any, 133
warn_unreachable, 133
warn_unused_configs, 140
warn_unused_ignores, 133
xml_report, 139
custom_typeshed_dir
configuration value, 138
custom_typing_module
configuration value, 138
D
disable_error_code
configuration value, 135
disallow_any_decorated
configuration value, 131
disallow_any_explicit
configuration value, 131
disallow_any_expr
configuration value, 131
disallow_any_generics
configuration value, 132
disallow_any_unimported
configuration value, 132
disallow_incomplete_defs
configuration value, 131
disallow_subclassing_any
configuration value, 132
disallow_untyped_calls
configuration value, 132
disallow_untyped_decorators
configuration value, 132
disallow_untyped_defs
configuration value, 132
dmypy command line option
--callsites, 145
--export-types, 144
--flex-any, 145
--force-reload, 147
--fswatcher-dump-file, 144
--include-kind, 146
--include-object-attrs, 146
--include-span, 146
--json, 145
--limit, 146
--log-file, 143
--max-guesses, 145
--no-any, 145
--no-errors, 145
--perf-stats-file, 144
--remove, 144
--show, 146
--status-file, 143
--timeout, 143
--union-attrs, 146
--update, 144
--use-fixme, 145
--verbose, 146

E
enable_error_code
configuration value, 135
error_summary
configuration value, 136
exclude
configuration value, 128
explicit_package_bases
configuration value, 129

F
files
configuration value, 128
follow_imports
configuration value, 129
follow_imports_for_stubs
configuration value, 130

H
html_report / xslt_html_report
configuration value, 139

I
ignore_errors
configuration value, 134
ignore_missing_imports
configuration value, 129
implicit_reexport
configuration value, 135
incremental
configuration value, 137

J
junit_xml
configuration value, 140

L
lcount_report
configuration value, 139
linecoverage_report
configuration value, 139
lineprecision_report
configuration value, 139
local_partial_types
configuration value, 134

M
mypy command line option
-V, 114
--allow-redefinition, 119
--allow-untyped-globals, 119
--always-false, 116
--always-true, 116
--any-exprs-report, 124
--cache-dir, 123
--cache-fine-grained, 123
--check-untyped-defs, 117
--cobertura-xml-report, 124
--command, 114
--config-file, 114
--custom-typeshed-dir, 123
--custom-typing-module, 123
--disable-error-code, 121
--disallow-any-decorated, 117
--disallow-any-explicit, 117
--disallow-any-expr, 117
--disallow-any-generics, 117
--disallow-any-unimported, 117
--disallow-incomplete-defs, 117
--disallow-subclassing-any, 117
--disallow-untyped-calls, 117
--disallow-untyped-decorators, 118
--disallow-untyped-defs, 117
--disable-error-code, 121
--exclude, 114
--explicit-package-bases, 115
--fast-module-lookup, 116
--find-occurrences, 125
--follow-imports, 115
--help, 114
--html-report, 124
--ignore-missing-imports, 115
--install-types, 125
--junit-xml, 125
--linecount-report, 124
--linecoverage-report, 124
--lineprecision-report, 124
--local-partial-types, 120
--module, 113
--namespace-packages, 115

Index 205
--no-color-output, 122
--no-error-summary, 122
--no-implicit-optional, 118
--no-implicit-reexport, 120
--no-incremental, 123
--no-silence-site-packages, 116
--no-site-packages, 115
--no-strict-optional, 118
--no-warn-no-return, 118
--non-interactive, 125
--package, 113
--pdb, 123
--platform, 116
--pretty, 122
--python-executable, 115
--python-version, 116
--raise-exceptions, 123
--scripts-are-modules, 125
--shadow-file, 124
--show-absolute-path, 122
--show-column-numbers, 122
--show-error-codes, 122
--show-error-context, 122
--show-error-end, 122
--show-traceback, 123
--skip-cache-mtime-checks, 123
--skip-version-check, 123
--soft-error-limit, 122
--sqlite-cache, 123
--strict, 121
--strict-equality, 120
--tb, 123
--txt-report, 124
--verbose, 114
--version, 114
--warn-incomplete-stub, 123
--warn-redundant-casts, 118
--warn-return-any, 119
--warn-unreachable, 119
--warn-unused-configs, 114
--warn-unused-ignores, 118
--xml-report, 125
-c, 114
-h, 114
-m, 113
-p, 113
-v, 114
mypy_path
configuration value, 128

N
namespace_packages
configuration value, 129
no_implicit_optional

P
pdb
configuration value, 138
platform
configuration value, 131
plugins
configuration value, 138
pretty
configuration value, 136
Python Enhancement Proposals
PEP 3107, 4
PEP 420, 115, 129
PEP 484, 1, 13, 150, 152, 154
PEP 484#function-method-overloading, 77
PEP 484#the-type-of-class-objects, 33, 34
PEP 484#type-aliases, 73
PEP 492, 87
PEP 508, 44
PEP 526, 1, 7, 13
PEP 544, 44, 197
PEP 544#generic-protocols, 72
PEP 557, 190
PEP 561, 60, 112, 115, 116, 130, 147, 148
PEP 561#stub-only-packages, 148
PEP 563, 40
PEP 585, 43
PEP 604, 30, 41, 44
PEP 613, 31, 167
PEP 647, 56
python_executable
configuration value, 130
python_version
configuration value, 131

R
raise_exceptions
configuration value, 138

S
scripts_are_modules
configuration value, 140
show_absolute_path
configuration value, 136
show_column_numbers
configuration value, 136
show_error_codes
configuration value, 136
show_error_context
configuration value, 136
show_none_errors
  configuration value, 134
show_traceback
  configuration value, 138
skip_cache_mtime_checks
  configuration value, 137
skip_version_check
  configuration value, 137
sqlite_cache
  configuration value, 137
strict
  configuration value, 135
strict_concatenate
  configuration value, 135
strict_equality
  configuration value, 135
strict_optional
  configuration value, 133
stubgen command line option
  --doc-dir, 155
  --export-less, 156
  --help, 156
  --ignore-errors, 156
  --include-private, 156
  --module, 155
  --no-import, 155
  --output, 156
  --package, 155
  --parse-only, 155
  --quiet, 156
  --search-path, 156
  --verbose, 156
  -h, 156
  -m, 155
  -o, 156
  -p, 155
  -q, 156
  -v, 156
stubtest command line option
  --allowlist, 158
  --check-typeshed, 158
  --concise, 158
  --custom-typeshed-dir, 158
  --generate-allowlist, 158
  --help, 158
  --ignore-missing-stub, 158
  --ignore-positional-only, 158
  --ignore-unused-allowlist, 158
  --mypy-config-file, 158

V

verbosity
  configuration value, 140

W

warn_incomplete_stub
  configuration value, 138
warn_no_return
  configuration value, 133
warn_redundant_casts
  configuration value, 133
warn_return_any
  configuration value, 133
warn_unreachable
  configuration value, 133
warn_unused_configs
  configuration value, 140
warn_unused_ignores
  configuration value, 133

X

xml_report
  configuration value, 139

txt_report / xslt_txt_report
  configuration value, 139