Mypy Documentation

Release 2.1.0

Jukka

May 11, 2026

FIRST STEPS

Mypy is a static type checker for Python.

Type checkers help ensure that you're using variables and functions in your code correctly. With mypy, add type hints (**PEP 484**) to your Python programs, and mypy will warn you when you use those types incorrectly.

Python is a dynamic language, so usually you'll only see errors in your code when you attempt to run it. Mypy is a *static* checker, so it finds bugs in your programs without even running them!

Here is a small example to whet your appetite:

```
number = input("What is your favourite number?")
print("It is", number + 1)  # error: Unsupported operand types for + ("str" and "int")
```

Adding type hints for mypy does not interfere with the way your program would otherwise run. Think of type hints as similar to comments! You can always use the Python interpreter to run your code, even if mypy reports errors.

Mypy is designed with gradual typing in mind. This means you can add type hints to your code base slowly and that you can always fall back to dynamic typing when static typing is not convenient.

Mypy has a powerful and easy-to-use type system, supporting features such as type inference, generics, callable types, tuple types, union types, structural subtyping and more. Using mypy will make your programs easier to understand, debug, and maintain.

Note

Although mypy is production ready, there may be occasional changes that break backward compatibility. The mypy development team tries to minimize the impact of changes to user code. In case of a major breaking change, mypy's major version will be bumped.

CHAPTER

ONE

CONTENTS

1.1 Getting started

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

If you're looking for a quick intro, see the *mypy cheatsheet*.

If you're unfamiliar with the concepts of static and dynamic type checking, 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 3.10 or later to run. You can 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 also means that you are always free to ignore the errors mypy reports, if you so wish. You can always use the Python interpreter to run your code, even if mypy reports errors.

However, if you try directly running mypy on your existing Python code, it will most likely report little to no errors. This is a feature! It makes it easy to adopt mypy incrementally.

In order to get useful diagnostics from mypy, you must add *type annotations* to your code. See the section below for details.

1.1.2 Dynamic vs static typing

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

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

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

This is the case even if you misuse the function!

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

# These calls will fail when the program runs, but mypy does not report an error
# because "greeting" does not have type annotations.
greeting(123)
greeting(b"Alice")
```

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

```
# The "name: str" annotation says that the "name" argument should be a string
# The "-> str" annotation says that "greeting" will return a string
def greeting(name: str) -> str:
    return 'Hello ' + name
```

This function is now *statically typed*: mypy will use the provided type hints to detect incorrect use of the `greeting` function and incorrect use of variables within the `greeting` function. For example:

```
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"
greeting("World!")  # No error

def bad_greeting(name: str) -> str:
    return 'Hello ' * name  # Unsupported operand types for * ("str" and "str")
```

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. You can also get mypy to provide some limited checking of dynamically typed functions by using the `--check-untyped-defs` flag. See *The mypy command line* for more information on configuring mypy.

1.1.3 Strict mode and configuration

Mypy has a *strict mode* that enables a number of additional checks, like `--disallow-untyped-defs`.

If you run mypy with the `--strict` flag, you will basically never get a type related error at runtime without a corresponding mypy error, unless you explicitly circumvent mypy somehow.

However, this flag will probably be too aggressive if you are trying to add static types to a large, existing codebase. See *Using mypy with an existing codebase* for suggestions on how to handle that case.

Mypy is very configurable, so you can start with using `--strict` and toggle off individual checks. For instance, if you use many third party libraries that do not have types, `--ignore-missing-imports` may be useful. See *Introduce stricter options* for how to build up to `--strict`.

See *The mypy command line* and *The mypy configuration file* for a complete reference on configuration options.

1.1.4 More complex types

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

You can express this idea using `collections.abc.Iterable`:

```
from collections.abc import Iterable  # or "from typing import Iterable"

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

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 *subtype* of `T`. That is, `list[str]` is a subtype of `Iterable[str]`.

This also applies to inheritance, so if you have a class `Child` that inherits from `Parent`, then a value of type `Child` can be assigned to a variable of type `Parent`. For example, a `RuntimeError` instance can be passed to a function that is annotated as taking an `Exception`.

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 a union type. For example, `int` is a subtype of `int | str`:

```
def normalize_id(user_id: int | str) -> str:
    if isinstance(user_id, int):
        return f'user-{100_000 + user_id}'
    else:
        return user_id
```

The `typing` module contains many other useful types.

For a quick overview, look through the *mypy cheatsheet*.

For a detailed overview (including information on how to make your own generic types or your own type aliases), look through the *type system reference*.

Note

When adding types, the convention is to import types using the form `from typing import <name>` (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.

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

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

For more details, see *Type inference and type annotations*.

1.1.6 Types from libraries

Mypy can also understand how to work with types from libraries that you use.

For instance, mypy comes out of the box with an intimate knowledge of the Python standard library. For example, here is a function which uses the `Path` object from the pathlib standard library module:

```
from pathlib import Path

def load_template(template_path: Path, name: str) -> str:
    # Mypy knows that `template_path` has a `read_text` method that returns a str
    template = template_path.read_text()
    # ...so it understands this line type checks
    return template.replace('USERNAME', name)
```

If a third party library you use *declares support for type checking*, mypy will type check your use of that library based on the type hints it contains.

However, if the third party library does not have type hints, mypy will complain about missing type information.

```
prog.py:1: error: Library stubs not installed for "yaml"
prog.py:1: note: Hint: "python3 -m pip install types-PyYAML"
prog.py:2: error: Library stubs not installed for "requests"
prog.py:2: note: Hint: "python3 -m pip install types-requests"
...
```

In this case, you can provide mypy a different source of type information, by installing a *stub* package. A stub package is a package that contains type hints for another library, but no actual code.

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

Stubs packages for a distribution are often named `types-<distribution>`. Note that a distribution name may be different from the name of the package that you import. For example, `types-PyYAML` contains stubs for the `yaml` package.

For more discussion on strategies for handling errors about libraries without type information, refer to *Missing imports*.

For more information about stubs, see *Stub files*.

1.1.7 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.
- For general questions about Python typing, try posting at typing discussions.

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 Type hints cheat sheet

This document is a quick cheat sheet showing how to use type annotations for various common types in Python.

1.2.1 Variables

Technically many of the type annotations shown below are redundant, since mypy can usually infer the type of a variable from its value. See *Type inference and type annotations* for more details.

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

# You don't need to initialize a variable to annotate it
```

(continues on next page)

(continued from previous page)

```
a: int  # Ok (no value at runtime until assigned)

# Doing so can be useful in conditional branches
child: bool
if age < 18:
    child = True
else:
    child = False
```

1.2.2 Useful built-in types

```
# For most types, just use the name of the type in the annotation
# Note that mypy can usually infer the type of a variable from its value,
# so technically these annotations are redundant
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
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}

# For tuples of fixed size, we specify the types of all the elements
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)

# On Python 3.8 and earlier, the name of the collection type is
# capitalized, and the type is imported from the 'typing' module
from typing import List, Set, Dict, Tuple
x: List[int] = [1]
x: Set[int] = {6, 7}
x: Dict[str, float] = {"field": 2.0}
x: Tuple[int, str, float] = (3, "yes", 7.5)
x: Tuple[int, ...] = (1, 2, 3)

from typing import Union, Optional

# Use the | operator when something could be one of a few types
x: list[int | str] = [3, 5, "test", "fun"]
# Union is equivalent
x: list[Union[int, str]] = [3, 5, "test", "fun"]

# Use X | None for a value that could be None; Optional[X] is the same as X | None
x: str | None = "something" if some_condition() else None
```

(continues on next page)

(continued from previous page)

```
if x is not None:
    # Mypy understands x won't be None here because of the if-statement
    print(x.upper())
# If you know a value can never be None due to some logic that mypy doesn't
# understand, use an assert
assert x is not None
print(x.upper())
```

1.2.3 Functions

```
from collections.abc import Iterator, Callable
from typing import Union, Optional

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

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

# If a function does not return a value, use None as the return type
# Default value for an argument goes after the type annotation
def show(value: str, excitement: int = 10) -> None:
    print(value + "!" * excitement)

# Note that arguments without a type are dynamically typed (treated as Any)
# and that functions without any annotations are not checked
def untyped(x):
    x.anything() + 1 + "string"  # no errors

# This is how you annotate a callable (function) value
x: Callable[[int, float], float] = f
def register(callback: Callable[[str], int]) -> None: ...

# A generator function that yields ints is secretly just a function that
# returns an iterator of ints, so that's how we annotate it
def gen(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: str | list[str],
    sender: str,
    cc: list[str] | None,
    bcc: list[str] | None,
    subject: str = '',
    body: list[str] | None = None,
```

(continues on next page)

(continued from previous page)

```
) -> bool:
    ...

# Mypy understands positional-only and keyword-only arguments
# Positional-only arguments can also be marked by using a name starting with
# two underscores
def quux(x: int, /, *, y: int) -> None:
    pass

quux(3, y=5)  # Ok
quux(3, 5)  # error: Too many positional arguments for "quux"
quux(x=3, y=5)  # error: Unexpected keyword argument "x" for "quux"

# This says each positional arg and each keyword arg is a "str"
def call(self, *args: str, **kwargs: str) -> str:
    reveal_type(args)  # Revealed type is "tuple[str, ...]"
    reveal_type(kwargs)  # Revealed type is "dict[str, str]"
    request = make_request(*args, **kwargs)
    return self.do_api_query(request)
```

1.2.4 Classes

```
from typing import ClassVar

class BankAccount:
    # 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, account_name: str, initial_balance: int = 0) -> None:
        # mypy will infer the correct types for these instance variables
        # based on the types of the parameters.
        self.account_name = account_name
        self.balance = initial_balance

    # For instance methods, omit type for "self"
    def deposit(self, amount: int) -> None:
        self.balance += amount

    def withdraw(self, amount: int) -> None:
        self.balance -= amount

# User-defined classes are valid as types in annotations
account: BankAccount = BankAccount("Alice", 400)
def transfer(src: BankAccount, dst: BankAccount, amount: int) -> None:
    src.withdraw(amount)
    dst.deposit(amount)

# Functions that accept BankAccount also accept any subclass of BankAccount!
class AuditedBankAccount(BankAccount):
    # You can optionally declare instance variables in the class body
    audit_log: list[str]
```

(continues on next page)

(continued from previous page)

```
    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"Withdrew {amount}")
        self.balance -= amount

audited = AuditedBankAccount("Bob", 300)
transfer(audited, account, 100)  # type checks!

# You can use the ClassVar annotation to declare a class variable
class Car:
    seats: ClassVar[int] = 4
    passengers: ClassVar[list[str]]

# If you want dynamic attributes on your class, have it
# override "__setattr__" or "__getattr__"
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 = A()
a.foo = 42  # Works
a.bar = 'Ex-parrot'  # Fails type checking
```

1.2.5 When you're puzzled or when things are complicated

```
from typing import Union, Any, Optional, TYPE_CHECKING, 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"

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

# 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()
```

(continues on next page)

(continued from previous page)

```
# Mypy will let you do anything with x!
x.whatever() * x["you"] + x("want") - any(x) and all(x) is super  # no errors

# 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 add a comment explaining the issue.
x = confusing_function()  # type: ignore  # confusing_function won't return None here␣
↪because ...

# "cast" is a helper function that lets you override the inferred
# type of an expression. It's only for mypy -- there's no runtime check.
a = [4]
b = cast(list[int], a)  # Passes fine
c = cast(list[str], a)  # Passes fine despite being a lie (no runtime check)
reveal_type(c)  # Revealed type is "builtins.list[builtins.str]"
print(c)  # Still prints [4] ... the object is not changed or casted at runtime

# Use "TYPE_CHECKING" if you want to have code that mypy can see but will not
# be executed at runtime (or to have code that mypy can't see)
if TYPE_CHECKING:
    import json
else:
    import orjson as json  # mypy is unaware of this
```

In some cases type annotations can cause issues at runtime, see *Annotation issues at runtime* for dealing with this.

See *Silencing type errors* for details on how to silence errors.

1.2.6 Standard "duck types"

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

```
from collections.abc import Mapping, MutableMapping, Sequence, Iterable
# or 'from typing import ...' (required in Python 3.8)

# 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'  # mypy will complain about this line...
    return list(my_mapping.keys())

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

(continues on next page)

(continued from previous page)

```
def f(my_mapping: MutableMapping[int, str]) -> set[str]:
    my_mapping[5] = 'maybe'  # ...but mypy is OK with this.
    return set(my_mapping.values())

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

import sys
from typing import IO

# Use IO[str] or IO[bytes] for functions that should accept or return
# objects that come from an open() call (note that 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
```

You can even make your own duck types using *Protocols and structural subtyping*.

1.2.7 Forward references

```
# You may want to reference a class before it is defined.
# This is known as a "forward reference".
def f(foo: A) -> int:  # This will fail at runtime with 'A' is not defined
    ...

# However, if you add the following special import:
from __future__ import annotations
# It will work at runtime and type checking will succeed as long as there
# is a class of that name later on in the file
def f(foo: A) -> int:  # Ok
    ...

# Another option is to just put the type in quotes
def f(foo: 'A') -> int:  # Also ok
    ...

class A:
    # This can also come up if you need to reference a class in a type
    # annotation inside the definition of that class
    @classmethod
    def create(cls) -> A:
        ...
```

See *Class name forward references* for more details.

1.2.8 Decorators

Decorator functions can be expressed via generics. See *Declaring decorators* for more details. Example using Python 3.12 syntax:

```
from collections.abc import Callable
from typing import Any

def bare_decorator[F: Callable[..., Any]](func: F) -> F:
    ...

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

The same example using pre-3.12 syntax:

```
from collections.abc import Callable
from typing import Any, TypeVar

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

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

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

1.2.9 Coroutines and asyncio

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

```
import asyncio

# A coroutine is typed like a normal function
async def countdown(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 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.

1.3.1 Start small

If your codebase is large, pick a subset of your codebase (say, 5,000 to 50,000 lines) and get mypy to run successfully only on this subset at first, *before adding annotations*. This should be doable in a day or two. The sooner you get some form of mypy passing on your codebase, the sooner you benefit.

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.

We'll mention some tips for getting mypy passing on your codebase in various sections below.

1.3.2 Run mypy consistently and prevent regressions

Make sure all developers on your codebase run mypy the same way. One way to ensure this is adding a small script with your mypy invocation to your codebase, or adding your mypy invocation to existing tools you use to run tests, like `tox`.

- Make sure everyone runs mypy with the same options. Checking a mypy *configuration file* into your codebase is the easiest way to do this.
- Make sure everyone type checks the same set of files. See *Specifying code to be checked* for details.
- Make sure everyone runs mypy with the same version of mypy, for instance by pinning mypy with the rest of your dev requirements.

In particular, you'll want to make sure to run mypy as part of your Continuous Integration (CI) system as soon as possible. This will prevent new type errors from being introduced into your codebase.

A simple CI script could look something like this:

```
python3 -m pip install mypy==1.8
# Run your standardised mypy invocation, e.g.
mypy my_project
# This could also look like `scripts/run_mypy.sh`, `tox run -e mypy`, `make mypy`, etc
```

1.3.3 Ignoring errors from certain modules

By default mypy will follow imports in your code and try to check everything. This means even if you only pass in a few files to mypy, it may still process a large number of imported files. This could potentially result in lots of errors you don't want to deal with at the moment.

One way to deal with this is to ignore errors in modules you aren't yet ready to type check. The *`ignore_errors`* option is useful for this, for instance, if you aren't yet ready to deal with errors from `package_to_fix_later`:

```
[mypy-package_to_fix_later.*]
ignore_errors = True
```

You could even invert this, by setting `ignore_errors = True` in your global config section and only enabling error reporting with `ignore_errors = False` for the set of modules you are ready to type check.

The per-module configuration that mypy's configuration file allows can be extremely useful. Many configuration options can be enabled or disabled only for specific modules. In particular, you can also enable or disable various error codes on a per-module basis, see *Error codes*.

1.3.4 Fixing errors related to imports

A common class of error you will encounter is errors from mypy about modules that it can't find, that don't have types, or don't have stub files:

```
core/config.py:7: error: Cannot find implementation or library stub for module named
↪'frobnicate'
core/model.py:9: error: Cannot find implementation or library stub for module named 'acme
↪'
...
```

Sometimes these can be fixed by installing the relevant packages or stub libraries in the environment you're running `mypy` in.

See *Missing imports* for a complete reference on these errors and the ways in which you can fix them.

You'll likely find that you want to suppress all errors from importing a given module that doesn't have types. If you only import that module in one or two places, you can use `# type: ignore` comments. For example, here we ignore an error about a third-party module `frobnicate` that doesn't have stubs using `# type: ignore`:

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

But if you import the module in many places, this becomes unwieldy. In this case, we recommend using a *configuration file*. For example, to disable errors about importing `frobnicate` and `acme` everywhere in your codebase, use a config like this:

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

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

If you get a large number of errors, you may want to ignore all errors about missing imports, for instance by setting `--disable-error-code=import-untyped`. or setting `ignore_missing_imports` to true globally. This can hide errors later on, so we recommend avoiding this if possible.

Finally, mypy allows fine-grained control over specific import following behaviour. It's very easy to silently shoot yourself in the foot when playing around with these, so this should be a last resort. For more details, look *here*.

1.3.5 Prioritise annotating 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.

Mypy is designed to support gradual typing, i.e. letting you add annotations at your own pace, so 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.

1.3.6 Write annotations as you go

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.

1.3.7 Automate annotation of legacy code

There are tools for automatically adding draft annotations based on simple static analysis or on type profiles collected at runtime. Tools include MonkeyType, autotyping 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.

1.3.8 Introduce stricter options

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

An excellent goal to aim for is to have your codebase pass when run against `mypy --strict`. This basically ensures that you will never have a type related error without an explicit circumvention somewhere (such as a `# type: ignore` comment).

The following config is equivalent to `--strict` (as of mypy 1.0):

```
# Start off with these
warn_unused_configs = True
warn_redundant_casts = True
warn_unused_ignores = True

# Getting this passing should be easy
strict_equality = True

# Strongly recommend enabling this one as soon as you can
check_untyped_defs = True

# These shouldn't be too much additional work, but may be tricky to
# get passing if you use a lot of untyped libraries
disallow_subclassing_any = True
disallow_untyped_decorators = True
disallow_any_generics = True

# These next few are various gradations of forcing use of type annotations
disallow_untyped_calls = True
disallow_incomplete_defs = True
disallow_untyped_defs = True

# This one isn't too hard to get passing, but return on investment is lower
no_implicit_reexport = True

# This one can be tricky to get passing if you use a lot of untyped libraries
warn_return_any = True

# This one is a catch-all flag for the rest of strict checks that are technically
# correct but may not be practical
extra_checks = True
```

Note that you can also start with `--strict` and subtract, for instance:

```
strict = True
warn_return_any = False
```

Remember that many of these options can be enabled on a per-module basis. For instance, you may want to enable `disallow_untyped_defs` for modules which you've completed annotations for, in order to prevent new code from being added without annotations.

And if you want, it doesn't stop at `--strict`. Mypy has additional checks that are not part of `--strict` that can be useful. See the complete *The mypy command line* reference and *Error codes for optional checks*.

1.3.9 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.

1.4 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:

Type	Description
`int`	integer
`float`	floating point number
`bool`	boolean value (subclass of `int`)
`str`	text, sequence of unicode codepoints
`bytes`	8-bit string, sequence of byte values
`object`	an arbitrary object (`object` is the common base class)

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`:

Type	Description
`Any`	dynamically typed value with an arbitrary type

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

1.4.3 Generic types

Built-in collection type objects support indexing:

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

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.

See *Using generic builtins* for more details.

These legacy types defined in typing are also supported:

Type	Description
List[str]	list of str objects
Tuple[int, int]	tuple of two int objects (Tuple[()] is the empty tuple)
Tuple[int, ...]	tuple of an arbitrary number of int objects
Dict[str, int]	dictionary from str keys to int values
Type[C]	type object of C (C is a class/type variable/union of types)

1.5 Type inference and type annotations

1.5.1 Type inference

For most variables, if you do not explicitly specify its type, mypy will infer the correct type based on what is initially assigned to the variable.

```
# Mypy will infer the type of these variables, despite no annotations
i = 1
reveal_type(i)  # Revealed type is "builtins.int"
l = [1, 2]
reveal_type(l)  # Revealed type is "builtins.list[builtins.int]"
```

Note

Note that mypy will not use type inference in dynamically typed functions (those without a function type annotation) — every local variable type defaults to Any in such functions. For more details, see *Dynamically typed code*.

```
def untyped_function():
    i = 1
    reveal_type(i) # Revealed type is "Any"
                   # 'reveal_type' always outputs 'Any' in unchecked functions
```

1.5.2 Explicit types for variables

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

```
x: 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 int | str (this type means that the value can be either an int or a str).

The best way to think about this is that the type annotation sets the type of the variable, not the type of the expression. For instance, mypy will complain about the following code:

```
x: int | str = 1.1  # error: Incompatible types in assignment
                    # (expression has type "float", variable has type "int | str")
```

> **Note**
>
> To explicitly override the type of an expression you can use `cast(<type>, <expression>)`. See *Casts* for details.

Note that you can explicitly declare the type of a variable without giving it an initial value:

```
# We only unpack two values, so there's no right-hand side value
# for mypy to infer the type of "cs" from:
a, b, *cs = 1, 2  # error: Need type annotation for "cs"

rs: list[int]  # no assignment!
p, q, *rs = 1, 2  # OK
```

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:

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

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

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

1.5.4 Compatibility of container types

A quick note: container types can sometimes be unintuitive. We'll discuss this more in *Invariance vs covariance*. For example, the following program generates a mypy error, because mypy treats `list[int]` as incompatible with `list[object]`:

```
def f(l: list[object], k: list[int]) -> None:
    l = k  # error: Incompatible types in assignment
```

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

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

Other container types like `dict` and `set` behave similarly.

You can still run the above program; it prints `x`. This illustrates the fact that static types 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.

Mypy will take into account the type of the variable on the left-hand side of an assignment when inferring the type of the expression on the right-hand side. For example, the following will type check:

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

The value expression `[1, 2]` is type checked with the additional context that it is being assigned to a variable of type `list[object]`. This is used to infer the type of the *expression* as `list[object]`.

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

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

foo([])  # OK
```

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

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

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

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

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

1.5.6 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 in its latest update, the web framework you use can now take an integer argument to `run()`, which starts it on localhost on that port. Like so:

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

However, the devs forgot to update their type annotations for `run`, so mypy still thinks `run` only expects `str` types. 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 web framework yourself, you can temporarily disable type checking on that line, by adding a `# type: ignore`:

```
# 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:

```
# Starting app on http://localhost:8000
app.run(8000)  # type: ignore  # `run()` in v2.0 accepts an `int`, as a port
```

Type ignore error codes

By default, mypy displays an error code for each error:

```
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*.

Other ways to silence errors

You can get mypy to silence errors about a specific variable by dynamically typing it with `Any`. See *Dynamically typed code* for more information.

```
from typing import Any

def f(x: Any, y: str) -> None:
    x = 'hello'
    x += 1  # OK
```

You can ignore all mypy errors in a file by adding a `# mypy: ignore-errors` at the top of the file:

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

You can also specify per-module configuration options in your *The mypy configuration file*. For example:

```
# Don't report errors in the 'package_to_fix_later' package
[mypy-package_to_fix_later.*]
ignore_errors = True

# Disable specific error codes in the 'tests' package
# Also don't require type annotations
[mypy-tests.*]
disable_error_code = var-annotated, has-type
allow_untyped_defs = True

# Silence import errors from the 'library_missing_types' package
[mypy-library_missing_types.*]
ignore_missing_imports = True
```

Finally, adding a `@typing.no_type_check` decorator to a class, method or function causes mypy to avoid type checking that class, method or function and to treat it as not having any type annotations.

```
@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:

```
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.

This should not be confused with the `object` type, which represents the set of all values. Unlike `object`, `Any` introduces type unsafety — see *Any vs. object* for more.

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

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

Declared (and inferred) types are ignored (or *erased*) at runtime. They are basically treated as comments, and thus the above code does not generate a runtime error, even though `s` gets an `int` value when the program is run, while the declared type of `s` is actually `str`! You need to be careful with `Any` types, since they let you lie to mypy, and this could easily hide bugs.

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

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

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

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

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

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

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

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

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

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

Note

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

1.6.3 Tuple types

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

```
# 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:

```
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:

```
from collections.abc 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
```

Note

Import Callable[...] from typing instead of collections.abc if you use Python 3.8 or earlier.

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

```
from collections.abc import Callable

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

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

In situations where more precise or complex types of callbacks are necessary one can use flexible *callback protocols*. Lambdas are also supported. The lambda argument and return value types cannot be given explicitly; they are always inferred based on context using bidirectional type inference:

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

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

Callables can also be used against type objects, matching their `__init__` or `__new__` signature:

```
from collections.abc import Callable

class C:
    def __init__(self, app: str) -> None:
        pass

CallableType = Callable[[str], C]

def class_or_callable(arg: CallableType) -> None:
    inst = arg("my_app")
    reveal_type(inst)  # Revealed type is "C"
```

This is useful if you want `arg` to be either a `Callable` returning an instance of `C` or the type of `C` itself. This also works with *callback protocols*.

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 `T1 | ... | Tn` to construct a union type. For example, if an argument has type `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:

```
def f(x: 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.

Python 3.9 and older only partially support this syntax. Instead, you can use the legacy `Union[T1, ..., Tn]` type constructor. Example:

```
from typing import Union
```

(continues on next page)

(continued from previous page)

```
def f(x: Union[int, str]) -> None:
    ...
```

It is also possible to use the new syntax with versions of Python where it isn't supported by the runtime with some limitations, if you use `from __future__ import annotations` (see *Annotation issues at runtime*):

```
from __future__ import annotations

def f(x: int | str) -> None:   # OK on Python 3.7 and later
    ...
```

1.6.6 Optional types and the None type

You can use `T | None` to define a type variant that allows `None` values, such as `int | None`. This is called an *optional type*:

```
def strlen(s: str) -> int | None:
    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)
```

You can also use the `Optional` type modifier, such as `Optional[int]` (`Optional[X]` is the shorthand for `Union[X, None]`):

```
from typing import Optional

def strlen(s: str) -> Optional[int]:
    ...
```

Most operations will not be allowed on unguarded `None` or *optional* values (values with an optional type):

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

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

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

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

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

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

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

```
class Resource:
    path: str | None = None

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

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

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

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

```
class Resource:
    path: str | None = None
    ...
```

This also works for attributes defined within methods:

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

Often it's easier to not use any initial value for an attribute. This way you don't need to use an optional type and can avoid `assert ... is not None` checks. No initial value is needed if you annotate an attribute in the class body:

```
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 'int | None' because of the assignment below
    if i > 0:
```

(continues on next page)

(continued from previous page)

```
        n = i
    ...
```

Sometimes you may get the error “Cannot determine type of <something>”. In this case you should add an explicit `... | None` annotation.

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

The type `Optional[T]` *does not* mean a function parameter with a default value. It simply means that `None` is a valid argument value. This is a common confusion because `None` is a common default value for parameters, and parameters with default values are sometimes called *optional* parameters (or arguments).

1.6.7 Type aliases

In certain situations, type names may end up being long and painful to type, especially if they are used frequently:

```
def f() -> 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 (this is an *implicit type alias*):

```
AliasType = 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*.

Python 3.12 introduced the `type` statement for defining *explicit type aliases*. Explicit type aliases are unambiguous and can also improve readability by making the intent clear:

```
type AliasType = 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:
    ...
```

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.

Aliases defined using the `type` statement have these properties, which distinguish them from implicit type aliases:

- The definition may contain forward references without having to use string literal escaping, since it is evaluated lazily.
- The alias can be used in type annotations, type arguments, and casts, but it can't be used in contexts which require a class object. For example, it's not valid as a base class and it can't be used to construct instances.

There is also use an older syntax for defining explicit type aliases (**PEP 613**):

```
from typing import TypeAlias

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

1.6.8 Named tuples

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

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

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

```
from typing import NamedTuple

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

Python 3.6 introduced an alternative, class-based syntax for named tuples with types:

```
from typing import NamedTuple

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

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

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:

```
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.

1.6.9 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 `type[C]` (or, on Python 3.8 and lower, `typing.Type[C]`) where `C` is a class. In other words, when `C` is the name of a class, using `C` to annotate an argument declares that the argument is an instance of `C` (or of a subclass of `C`), but using `type[C]` as an argument annotation declares that the argument is a class object deriving from `C` (or `C` itself).

For example, assume the following classes:

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

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

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

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

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

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

How would we annotate this function? Without the ability to parameterize `type`, the best we could do would be:

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

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

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

However, using the `type[C]` syntax and a type variable with an upper bound (see *Type variables with upper bounds*) we can do better (Python 3.12 syntax):

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

Here is the example using the legacy syntax (Python 3.11 and earlier):

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

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

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

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

Note

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

For more details about `type[]` and `typing.Type[]`, see **PEP 484: The type of class objects**.

1.6.10 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:

```
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:

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

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

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

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

This is slightly different from using `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:

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

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:

```
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:

```
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:

```
from collections.abc import Callable
from typing import 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:

```
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
```

(continues on next page)

(continued from previous page)

```
        ...

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.

In order to ensure that your code remains correct when renaming methods, it can be helpful to explicitly mark a method as overriding a base method. This can be done with the `@override` decorator. `@override` can be imported from `typing` starting with Python 3.12 or from `typing_extensions` for use with older Python versions. If the base method is then renamed while the overriding method is not, mypy will show an error:

```
from typing import override

class Base:
    def f(self, x: int) -> None:
        ...
    def g_renamed(self, y: str) -> None:
        ...

class Derived1(Base):
    @override
    def f(self, x: int) -> None:   # OK
        ...

    @override
    def g(self, y: str) -> None:   # Error: no corresponding base method found
        ...
```

Note

Use *–enable-error-code explicit-override* to require that method overrides use the `@override` decorator. Emit an error if it is missing.

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

As always, relying on dynamically typed code can be unsafe. There is no runtime enforcement that the method override returns a value that is compatible with the original return type, since annotations have no effect at runtime:

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

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

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:

```
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):

```
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.

Mypy allows you to omit the body for an abstract method, but if you do so, it is unsafe to call such method via `super()`. For example:

```python
from abc import abstractmethod
class Base:
    @abstractmethod
    def foo(self) -> int: pass
    @abstractmethod
    def bar(self) -> int:
        return 0
class Sub(Base):
    def foo(self) -> int:
        return super().foo() + 1  # error: Call to abstract method "foo" of "Base"
                                  # with trivial body via super() is unsafe
    @abstractmethod
    def bar(self) -> int:
        return super().bar() + 1  # This is OK however.
```

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:
```

(continues on next page)

(continued from previous page)

```
        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:

- Use of string literal types or type comments
- Use of `typing.TYPE_CHECKING`
- Use of `from __future__ import annotations` (**PEP 563**)

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

1.8.1 String literal types and type comments

Mypy lets you add type annotations using the (now deprecated) `# type:` type comment syntax. These were required with Python versions older than 3.6, since they didn't support type annotations on variables. Example:

```
a = 1  # type: int

def f(x):  # type: (int) -> int
    return x + 1

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

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:

```
def f(a: list['A']) -> None: ...  # OK, prevents NameError since A is defined later
def g(n: 'int') -> None: ...      # Also 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. Future Python versions (potentially Python 3.14) will by default 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* not defined using the `type` statement;
- *type narrowing*;
- type definitions (see `TypeVar`, `NewType`, `NamedTuple`);
- base classes.

```
# base class example
from __future__ import annotations

class A(tuple['B', 'C']): ... # String literal types needed here
class B: ...
class C: ...
```

Warning

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:

```
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:

```
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:

```
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`:

```
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    import bar

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

File `bar.py`:

```
from foo import listify
```

(continues on next page)

(continued from previous page)

```
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:

```
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*:

```
from typing import TYPE_CHECKING
from queue import Queue

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

class Tasks(BaseQueue):  # OK
    ...

task_queue: Tasks
reveal_type(task_queue.get())  # Reveals str
```

If your subclass is also generic, you can use the following (using the legacy syntax for generic classes):

```
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 and later, 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. You may still encounter issues (even if you use a recent Python version) when subclassing generic classes defined in third-party libraries if types are generic only in stubs.

1.8.7 Using types defined in stubs but not at runtime

Sometimes stubs that you're using may define types you wish to reuse 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*:

```
from __future__ import annotations
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from _typeshed import SupportsRichComparison

 def f(x: SupportsRichComparison) -> None
```

The `from __future__ import annotations` is required to avoid a `NameError` when using the imported symbol. For more information and caveats, see the section on *future annotations*.

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`:

```
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**), 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.

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

```
from typing_extensions import TypeIs
```

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

```
import sys
if sys.version_info >= (3, 13):
    from typing import TypeIs
else:
    from typing_extensions import TypeIs
```

This plays nicely well with following **PEP 508** dependency specification: `typing_extensions; python_version<"3.13"`

1.9 Protocols and structural subtyping

The Python type system supports two ways of deciding whether two objects are compatible as types: nominal subtyping and structural subtyping.

Nominal subtyping is strictly based on the class hierarchy. If class `Dog` inherits class `Animal`, it's a subtype of `Animal`. Instances of `Dog` can be used when `Animal` instances are expected. This form of subtyping is what Python's type system predominantly uses: it's easy to understand and produces clear and concise error messages, and matches how the native `isinstance` check works – based on class hierarchy.

Structural subtyping is based on the operations that can be performed with an object. Class `Dog` is a structural subtype of class `Animal` 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. See **PEP 544** for the detailed specification of protocols and structural subtyping in Python.

1.9.1 Predefined protocols

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

```
from __future__ import annotations

from collections.abc import Iterator, Iterable

class IntList:
    def __init__(self, value: int, next: IntList | None) -> 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)
```

(continues on next page)

(continued from previous page)

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

Predefined protocol reference lists various protocols defined in `collections.abc` and `typing` and the signatures of the corresponding methods you need to define to implement each protocol.

Note

`typing` also contains deprecated aliases to protocols and ABCs defined in `collections.abc`, such as `Iterable[T]`.

1.9.2 Simple user-defined protocols

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

```
from collections.abc import Iterable
from typing import Protocol

class SupportsClose(Protocol):
    # Empty method body (explicit '...')
    def close(self) -> None: ...

class Resource:  # No SupportsClose base class!

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

    # ... other methods ...

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

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

`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()`.

1.9.3 Defining subprotocols and subclassing protocols

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

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

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

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

(continues on next page)

(continued from previous 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. In particular, omitting a value for an attribute or a method body will make it implicitly abstract:

```
class SomeProto(Protocol):
    attr: int  # Note, no right hand side
    def method(self) -> str: ...  # Literally just ... here

class ExplicitSubclass(SomeProto):
    pass

ExplicitSubclass()  # error: Cannot instantiate abstract class 'ExplicitSubclass'
                    # with abstract attributes 'attr' and 'method'
```

Similarly, explicitly assigning to a protocol instance can be a way to ask the type checker to verify that your class implements a protocol:

```
_proto: SomeProto = cast(ExplicitSubclass, None)
```

1.9.4 Invariance of protocol attributes

A common issue with protocols is that protocol attributes are invariant. For example:

```
class Box(Protocol):
      content: object

class IntBox:
      content: int

def takes_box(box: Box) -> None: ...

takes_box(IntBox())  # error: Argument 1 to "takes_box" has incompatible type "IntBox";␣
↪expected "Box"
                     # note:  Following member(s) of "IntBox" have conflicts:
                     # note:      content: expected "object", got "int"
```

This is because Box defines content as a mutable attribute. Here's why this is problematic:

```
def takes_box_evil(box: Box) -> None:
    box.content = "asdf"  # This is bad, since box.content is supposed to be an object

my_int_box = IntBox()
takes_box_evil(my_int_box)
my_int_box.content + 1  # Oops, TypeError!
```

This can be fixed by declaring content to be read-only in the Box protocol using @property:

```
class Box(Protocol):
    @property
    def content(self) -> object: ...

class IntBox:
    content: int

def takes_box(box: Box) -> None: ...

takes_box(IntBox(42))  # OK
```

1.9.5 Recursive protocols

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

```
from __future__ import annotations

from typing import Protocol

class TreeLike(Protocol):
    value: int

    @property
    def left(self) -> TreeLike | None: ...
```

(continues on next page)

(continued from previous page)

```
    @property
    def right(self) -> TreeLike | None: ...

class SimpleTree:
    def __init__(self, value: int) -> None:
        self.value = value
        self.left: SimpleTree | None = None
        self.right: SimpleTree | None = None

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

1.9.6 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 rudimentary support for runtime structural checks:

```
from typing 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):  # Works at runtime!
   use(mug.handles)
```

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

> **Warning**
>
> `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 exist, not that they have the correct type. `issubclass()` with protocols will only check for the existence of methods.

Note

isinstance() with protocols can also be surprisingly slow. In many cases, you're better served by using hasattr() to check for the presence of attributes.

1.9.7 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 collections.abc import Iterable
from typing import Optional, Protocol

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

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

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

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

Callback protocols and Callable types can be used mostly interchangeably. Parameter names in __call__ methods must be identical, unless the parameters are positional-only. Example (using the legacy syntax for generic functions):

```
from collections.abc import Callable
from typing import Protocol, TypeVar

T = TypeVar('T')

class Copy(Protocol):
    # '/' marks the end of positional-only parameters
    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
```

1.9.8 Binding of types in protocol attributes

All protocol attributes annotations are treated as externally visible types of those attributes. This means that for example callables are not bound, and descriptors are not invoked:

```
from typing import Callable, Protocol, overload

class Integer:
    @overload
    def __get__(self, instance: None, owner: object) -> Integer: ...
    @overload
    def __get__(self, instance: object, owner: object) -> int: ...
    # <some implementation>

class Example(Protocol):
    foo: Callable[[object], int]
    bar: Integer

ex: Example
reveal_type(ex.foo)  # Revealed type is Callable[[object], int]
reveal_type(ex.bar)  # Revealed type is Integer
```

In other words, protocol attribute types are handled as they would appear in a `self` attribute annotation in a regular class. If you want some protocol attributes to be handled as though they were defined at class level, you should declare them explicitly using `ClassVar[...]`. Continuing previous example:

```
from typing import ClassVar

class OtherExample(Protocol):
    # This style is *not recommended*, but may be needed to reuse
    # some complex callable types. Otherwise use regular methods.
    foo: ClassVar[Callable[[object], int]]
    # This may be needed to mimic descriptor access on Type[...] types,
    # otherwise use a plain "bar: int" style.
    bar: ClassVar[Integer]

ex2: OtherExample
reveal_type(ex2.foo)  # Revealed type is Callable[[], int]
reveal_type(ex2.bar)  # Revealed type is int
```

1.9.9 Predefined protocol reference

Iteration protocols

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

collections.abc.Iterable[T]

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

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

See also `Iterable`.

collections.abc.Iterator[T]

```
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.

collections.abc.Sized

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

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

See also Sized.

collections.abc.Container[T]

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

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

See also Container.

collections.abc.Collection[T]

```
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.

collections.abc.Reversible[T]

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

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

See also Reversible.

typing.SupportsAbs[T]

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

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

See also SupportsAbs.

typing.SupportsBytes

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

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

See also `SupportsBytes`.

typing.SupportsComplex

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

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

See also `SupportsComplex`.

typing.SupportsFloat

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

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

See also `SupportsFloat`.

typing.SupportsInt

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

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

See also `SupportsInt`.

typing.SupportsRound[T]

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

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

See also `SupportsRound`.

Async protocols

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

collections.abc.Awaitable[T]

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

See also `Awaitable`.

collections.abc.AsyncIterable[T]

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

See also `AsyncIterable`.

collections.abc.AsyncIterator[T]

```
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.

contextlib.AbstractContextManager[T]

```
def __enter__(self) -> T
def __exit__(self,
             exc_type: type[BaseException] | None,
             exc_value: BaseException | None,
             traceback: TracebackType | None) -> bool | None
```

See also AbstractContextManager.

contextlib.AbstractAsyncContextManager[T]

```
def __aenter__(self) -> Awaitable[T]
def __aexit__(self,
              exc_type: type[BaseException] | None,
              exc_value: BaseException | None,
              traceback: TracebackType | None) -> Awaitable[bool | None]
```

See also AbstractAsyncContextManager.

1.10 Dynamically typed code

In *Dynamic vs static typing*, we discussed how bodies of functions that don't have any explicit type annotations in their function are "dynamically typed" and that mypy will not check them. In this section, we'll talk a little bit more about what that means and how you can enable dynamic typing on a more fine grained basis.

In cases where your code is too magical for mypy to understand, you can make a variable or parameter dynamically typed by explicitly giving it the type `Any`. Mypy will let you do basically anything with a value of type `Any`, including assigning a value of type `Any` to a variable of any type (or vice versa).

```
from typing import Any

num = 1         # Statically typed (inferred to be int)
num = 'x'       # error: Incompatible types in assignment (expression has type "str",␣
↪variable has type "int")

dyn: Any = 1    # Dynamically typed (type Any)
dyn = 'x'       # OK

num = dyn       # No error, mypy will let you assign a value of type Any to any variable
num += 1        # Oops, mypy still thinks num is an int
```

You can think of Any as a way to locally disable type checking. See *Silencing type errors* for other ways you can shut up the type checker.

1.10.1 Operations on Any values

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

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

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

```
def f(x: Any) -> None:
    y = x.foo()
    reveal_type(y)  # Revealed type is "Any"
    z = y.bar("mypy will let you do anything to y")
    reveal_type(z)  # Revealed type is "Any"
```

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

Function parameters without annotations are also implicitly Any:

```
def f(x) -> None:
    reveal_type(x)  # Revealed type is "Any"
    x.can.do["anything", x]("wants", 2)
```

You can make mypy warn you about untyped function parameters using the *--disallow-untyped-defs* flag.

Generic types missing type parameters will have those parameters implicitly treated as Any:

```
def f(x: list) -> None:
    reveal_type(x)        # Revealed type is "builtins.list[Any]"
    reveal_type(x[0])     # Revealed type is "Any"
    x[0].anything_goes()  # OK
```

You can make mypy warn you about missing generic parameters using the *--disallow-any-generics* flag.

Finally, another major source of Any types leaking into your program is from third party libraries that mypy does not know about. This is particularly the case when using the *--ignore-missing-imports* flag. See *Missing imports* for more information about this.

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:

```
def f(o: object) -> None:
    if o:
```

(continues on next page)

(continued from previous page)

```
        print(o)
    print(isinstance(o, int))
    o = 2
    o = 'foo'
```

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

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

If you're not sure whether you need to use `object` or `Any`, use `object` – only switch to using `Any` if you get a type checker complaint.

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`.

The following type narrowing techniques are available:

- *Type narrowing expressions*
- *Casts*
- *User-Defined Type Guards*
- *TypeIs*

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
- `obj is not None` will narrow object to its *non-optional form*

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

```
def function(arg: object):
    if isinstance(arg, int):
        # Type is narrowed within the ``if`` branch only
        reveal_type(arg)  # Revealed type: "builtins.int"
```

(continues on next page)

(continued from previous page)

```
    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:

```
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:

```
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.

```
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.

```
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:

```
class MyCalcMeta(type):
    @classmethod
```

(continues on next page)

(continued from previous page)

```
    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:

```
from collections.abc 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`
```

The `callable` function can even split union types into callable and non-callable parts:

```
from collections.abc import Callable

x: 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"
```

1.11.2 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:

```
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:

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

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

```
from typing import cast, Any

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

1.11.3 User-Defined Type Guards

Mypy supports User-Defined Type Guards (**PEP 647**).

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:

```
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

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 (Python 3.12 syntax):

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

def is_two_element_tuple[T](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 3.12 syntax):

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

def is_set_of[T](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 a `TypeGuard`:

```
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:
>
> ```
> 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.

```
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.

1.11.4 TypeIs

Mypy supports TypeIs (**PEP 742**).

A TypeIs narrowing function allows you to define custom type checks that can narrow the type of a variable in both the if and else branches of a conditional, similar to how the built-in isinstance() function works.

TypeIs is new in Python 3.13 — for use in older Python versions, use the backport from typing_extensions

Consider the following example using TypeIs:

```
from typing import TypeIs

def is_str(x: object) -> TypeIs[str]:
    return isinstance(x, str)

def process(x: int | str) -> None:
    if is_str(x):
        reveal_type(x)  # Revealed type is 'str'
        print(x.upper())  # Valid: x is str
    else:
        reveal_type(x)  # Revealed type is 'int'
        print(x + 1)  # Valid: x is int
```

In this example, the function is_str is a type narrowing function that returns TypeIs[str]. When used in an if statement, x is narrowed to str in the if branch and to int in the else branch.

Key points:

- The function must accept at least one positional argument.
- The return type is annotated as `TypeIs[T]`, where `T` is the type you want to narrow to.
- The function must return a `bool` value.
- In the `if` branch (when the function returns `True`), the type of the argument is narrowed to the intersection of its original type and `T`.
- In the `else` branch (when the function returns `False`), the type of the argument is narrowed to the intersection of its original type and the complement of `T`.

TypeIs vs TypeGuard

While both TypeIs and TypeGuard allow you to define custom type narrowing functions, they differ in important ways:

- **Type narrowing behavior**: TypeIs narrows the type in both the if and else branches, whereas TypeGuard narrows only in the if branch.
- **Compatibility requirement**: TypeIs requires that the narrowed type T be compatible with the input type of the function. TypeGuard does not have this restriction.
- **Type inference**: With TypeIs, the type checker may infer a more precise type by combining existing type information with T.

Here's an example demonstrating the behavior with TypeGuard:

```
from typing import TypeGuard, reveal_type

def is_str(x: object) -> TypeGuard[str]:
    return isinstance(x, str)

def process(x: int | str) -> None:
    if is_str(x):
        reveal_type(x)  # Revealed type is "builtins.str"
        print(x.upper())  # ok: x is str
    else:
        reveal_type(x)  # Revealed type is "Union[builtins.int, builtins.str]"
```

(continues on next page)

(continued from previous page)

```
        print(x + 1)  # ERROR: Unsupported operand types for + ("str" and "int") ␣
↪[operator]
```

Generic TypeIs

TypeIs functions can also work with generic types:

```
from typing import TypeVar, TypeIs

T = TypeVar('T')

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

def process(names: tuple[str, ...]) -> None:
    if is_two_element_tuple(names):
        reveal_type(names)  # Revealed type is 'tuple[str, str]'
    else:
        reveal_type(names)  # Revealed type is 'tuple[str, ...]'
```

TypeIs with Additional Parameters

TypeIs functions can accept additional parameters beyond the first. The type narrowing applies only to the first argument.

```
from typing import Any, TypeVar, reveal_type, TypeIs

T = TypeVar('T')

def is_instance_of(val: Any, typ: type[T]) -> TypeIs[T]:
    return isinstance(val, typ)

def process(x: Any) -> None:
    if is_instance_of(x, int):
        reveal_type(x)  # Revealed type is 'int'
        print(x + 1)  # ok
    else:
        reveal_type(x)  # Revealed type is 'Any'
```

TypeIs in Methods

A method can also serve as a `TypeIs` function. Note that in instance or class methods, the type narrowing applies to the second parameter (after `self` or `cls`).

```
class Validator:
    def is_valid(self, instance: object) -> TypeIs[str]:
        return isinstance(instance, str)

    def process(self, to_validate: object) -> None:
        if Validator().is_valid(to_validate):
            reveal_type(to_validate)  # Revealed type is 'str'
            print(to_validate.upper())  # ok: to_validate is str
```

Assignment Expressions with TypeIs

You can use the assignment expression operator `:=` with `TypeIs` to create a new variable and narrow its type simultaneously.

```
from typing import TypeIs, reveal_type

def is_float(x: object) -> TypeIs[float]:
    return isinstance(x, float)

def main(a: object) -> None:
    if is_float(x := a):
        reveal_type(x)  # Revealed type is 'float'
        # x is narrowed to float in this block
        print(x + 1.0)
```

1.11.5 Limitations

Mypy's analysis is limited to individual symbols and it will not track relationships between symbols. For example, in the following code it's easy to deduce that if `a` is None then `b` must not be, therefore `a or b` will always be an instance of `C`, but Mypy will not be able to tell that:

```
class C:
    pass

def f(a: C | None, b: C | None) -> C:
    if a is not None or b is not None:
        return a or b  # Incompatible return value type (got "C | None", expected "C")
    return C()
```

Tracking these sort of cross-variable conditions in a type checker would add significant complexity and performance overhead.

You can use an `assert` to convince the type checker, override it with a *cast* or rewrite the function to be slightly more verbose:

```
def f(a: C | None, b: C | None) -> C:
    if a is not None:
        return a
    elif b is not None:
        return b
    return C()
```

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`.

> **Note**
>
> bytearray and memoryview were duck type compatible with bytes by default prior to mypy 2.0. This can still be enabled with *--no-strict-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:

```
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

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 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.

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:

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

# Function bodies cannot be completely removed. By convention,
# we replace them with `...` instead of the `pass` statement.
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:

```
from typing import Protocol

class Resource(Protocol):
    def ok_1(self, foo: list[str] = ...) -> None: ...

    def ok_2(self, foo: list[str] = ...) -> None:
        raise NotImplementedError()
```

(continues on next page)

(continued from previous page)

```
    def ok_3(self, foo: list[str] = ...) -> None:
        """Some docstring"""
        pass

    # Error: Incompatible default for parameter "foo" (default has
    # type "ellipsis", parameter 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 arguments, similar to built-in types such as `list[T]`. 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 accept one or more type arguments within `[...]`, which can be arbitrary types. For example, the type `dict[int, str]` has the type arguments `int` and `str`, and `list[int]` has the type argument `int`.

Programs can also define new generic classes. Here is a very simple generic class that represents a stack (using the syntax introduced in Python 3.12):

```
class Stack[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
```

There are two syntax variants for defining generic classes in Python. Python 3.12 introduced a new dedicated syntax for defining generic classes (and also functions and type aliases, which we will discuss later). The above example used the new syntax. Most examples are given using both the new and the old (or legacy) syntax variants. Unless mentioned otherwise, they work the same – but the new syntax is more readable and more convenient.

Here is the same example using the old syntax (required for Python 3.11 and earlier, but also supported on newer Python versions):

```
from typing import TypeVar, Generic

T = TypeVar('T')  # Define type variable "T"

class Stack(Generic[T]):
    def __init__(self) -> None:
```

(continues on next page)

(continued from previous page)

```
        # 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
```

Note

There are currently no plans to deprecate the legacy syntax. You can freely mix code using the new and old syntax variants, even within a single file (but *not* within a single class).

The `Stack` class can be used to represent a stack of any type: `Stack[int]`, `Stack[tuple[int, str]]`, etc. You can think of `Stack[int]` as referring to the definition of `Stack` above, but with all instances of `T` replaced with `int`.

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

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

# error: Argument 1 to "push" of "Stack" has incompatible type "str"; expected "int"
stack.push('x')

stack2: Stack[str] = Stack()
stack2.push('x')
```

Construction of instances of generic types is type checked (Python 3.12 syntax):

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

Box(1)       # OK, inferred type is Box[int]
Box[int](1)  # Also OK

# error: Argument 1 to "Box" has incompatible type "str"; expected "int"
Box[int]('some string')
```

Here is the definition of `Box` using the legacy syntax (Python 3.11 and earlier):

```
from typing import TypeVar, Generic

T = TypeVar('T')

class Box(Generic[T]):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, content: T) -> None:
        self.content = content
```

Note

Before moving on, let's clarify some terminology. The name `T` in `class Stack[T]` or `class Stack(Generic[T])` declares a *type parameter* `T` (of class `Stack`). `T` is also called a *type variable*, especially in a type annotation, such as in the signature of `push` above. When the type `Stack[...]` is used in a type annotation, the type within square brackets is called a *type argument*. This is similar to the distinction between function parameters and arguments.

1.14.2 Defining subclasses of generic classes

User-defined generic classes and generic classes defined in `typing` can be used as a base class for another class (generic or non-generic). For example (Python 3.12 syntax):

```
from typing import Mapping, Iterator

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

items: MyMap[str, int]  # OK

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

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

# This is a user-defined generic class
class Receiver[T]:
    def accept(self, value: T) -> None: ...

# This is a generic subclass of Receiver
class AdvancedReceiver[T](Receiver[T]): ...
```

Here is the above example using the legacy syntax (Python 3.11 and earlier):

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

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

# This is a generic subclass of Mapping
class MyMap(Mapping[KT, VT]):
    def __getitem__(self, k: KT) -> VT: ...
```

(continues on next page)

(continued from previous page)

```
    def __iter__(self) -> Iterator[KT]: ...
    def __len__(self) -> int: ...

items: MyMap[str, int]  # OK

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

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

# This is a user-defined generic class
class Receiver(Generic[T]):
    def accept(self, value: T) -> None: ...

# This is a generic subclass of Receiver
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*.

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

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

Example:

```
from typing import Generic, TypeVar, Any

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

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

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

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
```

When using the Python 3.12 syntax, all type parameters must always be explicitly defined immediately after the class name within `[...]`, and the `Generic[...]` base class is never used.

1.14.3 Generic functions

Functions can also be generic, i.e. they can have type parameters (Python 3.12 syntax):

```
from collections.abc import Sequence

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

Here is the same example using the legacy syntax (Python 3.11 and earlier):

```
from typing import TypeVar, Sequence

T = TypeVar('T')

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

As with generic classes, the type parameter `T` can be replaced with any type. That means `first` can be passed an argument with any sequence type, and the return type is derived from the sequence item type. Example:

```
reveal_type(first([1, 2, 3]))   # Revealed type is "builtins.int"
reveal_type(first(('a', 'b')))  # Revealed type is "builtins.str"
```

When using the legacy syntax, 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 to declare type parameters:

```
from typing import TypeVar, Sequence

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

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

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

Since the Python 3.12 syntax is more concise, it doesn't need (or have) an equivalent way of sharing type parameter definitions.

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

When calling a generic function, you can't explicitly pass the values of type parameters as type arguments. The values of type parameters are always inferred by mypy. This is not valid:

```
first[int]([1, 2])  # Error: can't use [...] with generic function
```

If you really need this, you can define a generic class with a `__call__` method.

1.14.4 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 it is specified using `T: <bound>` when using the Python 3.12 syntax. In the definition of a generic function or a generic class that uses such a type variable `T`, the type represented by `T` is assumed to be a subtype of its upper bound, so you can use methods of the upper bound on values of type `T` (Python 3.12 syntax):

```
from typing import SupportsAbs

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

An upper bound can also be specified with the `bound=...` keyword argument to `TypeVar`. Here is the example using the legacy syntax (Python 3.11 and earlier):

```
from typing import TypeVar, SupportsAbs

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

def max_by_abs(*xs: T) -> T:
    return max(xs, key=abs)
```

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:

```
max_by_abs(-3.5, 2)   # Okay, has type 'float'
max_by_abs(5+6j, 7)   # Okay, has type 'complex'
max_by_abs('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.

1.14.5 Generic methods and generic self

You can also define generic methods. In particular, the `self` parameter may also be generic, allowing a method to return the most precise type known at the point of access. In this way, for example, you can type check a chain of setter methods (Python 3.12 syntax):

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

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

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

(continues on next page)

(continued from previous page)

```
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, since the return type of `set_scale` would be `Shape`, which doesn't define `set_radius` or `set_width`.

When using the legacy syntax, just use a type variable in the method signature that is different from class type parameters (if any are defined). Here is the above example using the legacy syntax (3.11 and earlier):

```
from typing import TypeVar

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

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

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

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

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

Other uses include factory methods, such as copy and deserialization methods. For class methods, you can also define generic `cls`, using `type[T]` or `Type[T]` (Python 3.12 syntax):

```
class Friend:
    other: "Friend | None" = None

    @classmethod
    def make_pair[T: Friend](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()
```

Here is the same example using the legacy syntax (3.11 and earlier):

```
from typing import TypeVar

```

(continues on next page)

(continued from previous page)

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

class Friend:
    other: "Friend | None" = 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 or a `# type: ignore` comment.

Mypy lets you use generic self types in certain unsafe ways in order to support common idioms. For example, using a generic self type in an argument type is accepted even though it's unsafe (Python 3.12 syntax):

```
class Base:
    def compare[T: Base](self: T, other: T) -> bool:
        return False

class Sub(Base):
    def __init__(self, x: int) -> None:
        self.x = x

    # This is unsafe (see below) but allowed because it's
    # a common pattern and rarely causes issues in practice.
    def compare(self, other: 'Sub') -> bool:
        return self.x > other.x

b: Base = Sub(42)
b.compare(Base())  # Runtime error here: 'Base' object has no attribute 'x'
```

For some advanced uses of self types, see *additional examples*.

1.14.6 Automatic self types using typing.Self

Since the patterns described above are quite common, mypy supports a simpler syntax, introduced in **PEP 673**, to make them easier to use. Instead of introducing a type parameter and using an explicit annotation for `self`, you can import the special type `typing.Self` that is automatically transformed into a method-level type parameter with the current class as the upper bound, and you don't need an annotation for `self` (or `cls` in class methods). The example from the previous section can be made simpler by using `Self`:

```python
from typing import Self

class Friend:
    other: Self | None = None

    @classmethod
    def make_pair(cls) -> tuple[Self, Self]:
        a, b = cls(), cls()
        a.other = b
        b.other = a
        return a, b

class SuperFriend(Friend):
    pass

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

This is more compact than using explicit type parameters. Also, you can use `Self` in attribute annotations in addition to methods.

> **Note**
>
> To use this feature on Python versions earlier than 3.11, you will need to import `Self` from `typing_extensions` (version 4.0 or newer).

1.14.7 Variance of generic types

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

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

Let us illustrate this by few simple examples:

```python
# We'll use these classes in the examples below
class Shape: ...
class Triangle(Shape): ...
class Square(Shape): ...
```

- Most immutable container types, such as `Sequence` and `frozenset` are covariant. Union types are also covariant in all union items: `Triangle | int` is a subtype of `Shape | int`.

  ```python
  def count_lines(shapes: Sequence[Shape]) -> int:
      return sum(shape.num_sides for shape in shapes)

  triangles: Sequence[Triangle]
  count_lines(triangles)  # OK
  ```

(continues on next page)

(continued from previous page)

```
def foo(triangle: Triangle, num: int) -> None:
    shape_or_number: Union[Shape, int]
    # a Triangle is a Shape, and a Shape is a valid Union[Shape, int]
    shape_or_number = triangle
```

Covariance should feel relatively intuitive, but contravariance and invariance can be harder to reason about.

- `Callable` is an example of type that behaves contravariant in types of arguments. That is, `Callable[[Shape], int]` is a subtype of `Callable[[Triangle], int]`, despite `Shape` being a supertype of `Triangle`. To understand this, consider:

```
def cost_of_paint_required(
    triangle: Triangle,
    area_calculator: Callable[[Triangle], float]
) -> float:
    return area_calculator(triangle) * DOLLAR_PER_SQ_FT

# This straightforwardly works
def area_of_triangle(triangle: Triangle) -> float: ...
cost_of_paint_required(triangle, area_of_triangle)  # OK

# But this works as well!
def area_of_any_shape(shape: Shape) -> float: ...
cost_of_paint_required(triangle, area_of_any_shape)  # OK
```

`cost_of_paint_required` needs a callable that can calculate the area of a triangle. If we give it a callable that can calculate the area of an arbitrary shape (not just triangles), everything still works.

- `list` is an invariant generic type. Naively, one would think that it is covariant, like `Sequence` above, but consider this code:

```
class Circle(Shape):
    # The rotate method is only defined on Circle, not on Shape
    def rotate(self): ...

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

my_circles: list[Circle] = []
add_one(my_circles)     # This may appear safe, but...
my_circles[0].rotate()  # ...this will fail, since my_circles[0] is now a Shape,
↪not a Circle
```

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

When using the Python 3.12 syntax for generics, mypy will automatically infer the most flexible variance for each class type variable. Here `Box` will be inferred as covariant:

```
class Box[T]:  # this type is implicitly covariant
    def __init__(self, content: T) -> None:
        self._content = content

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

(continues on next page)

(continued from previous page)

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

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

Here the underscore prefix for `_content` is significant. Without an underscore prefix, the class would be invariant, as the attribute would be understood as a public, mutable attribute (a single underscore prefix has no special significance for mypy in most other contexts). By declaring the attribute as `Final`, the class could still be made covariant:

```
from typing import Final

class Box[T]:  # this type is implicitly covariant
    def __init__(self, content: T) -> None:
        self.content: Final = content

    def get_content(self) -> T:
        return self.content
```

When using the legacy syntax, mypy assumes that all user-defined generics are invariant by default. To declare a given generic class as covariant or contravariant, use type variables defined with special keyword arguments `covariant` or `contravariant`. For example (Python 3.11 or earlier):

```
from typing import Generic, TypeVar

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

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

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

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

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

1.14.8 Value-constrained type variables

By default, a type variable can be replaced with any type – or any type that is a subtype of the upper bound, which defaults to `object`. 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`. This lets us define a function that can concatenate two strings or bytes objects, but it can't be called with other argument types (Python 3.12 syntax):

```
def concat[S: (str, bytes)](x: S, y: S) -> S:
    return x + y

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

The same thing is also possibly using the legacy syntax (Python 3.11 or earlier):

```
from typing import TypeVar

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

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

No matter which syntax you use, such a type variable is called a value-constrained type variable (the allowed types are accessible at runtime via `TypeVar.__constraints__`). Importantly, this is different from a union type, since combinations of `str` and `bytes` are not accepted:

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

In this case, this is exactly what we want, since it's not possible to concatenate a string and a bytes object! If we tried to use a union type, the type checker would complain about this possibility:

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

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

```
class S(str): pass

ss = concat(S('foo'), S('bar'))
reveal_type(ss)  # Revealed type is "builtins.str"
```

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 using `str | bytes` as an upper bound, where the return type would be `S` (see *Type variables with upper bounds*). Using a value constraint is correct for `concat`, since `concat` actually returns a `str` instance in the above example:

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

You can also use type variables with a restricted set of possible values when defining a generic class. For example, the type `Pattern[S]` is used for the return value of `re.compile()`, where `S` can be either `str` or `bytes`. Regular expressions can be based on a string or a bytes pattern.

A type variable may not have both value constraints and an upper bound.

Note that you may come across `AnyStr` imported from `typing`. This feature is now deprecated, but it means the same as our definition of `AnyStr` above.

1.14.9 Declaring decorators

Decorators are typically functions that take a function as an argument and return another function. Describing this behaviour in terms of types can be a little tricky; we'll show how you can use type variables and a special kind of type variable called a *parameter specification* to do so.

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:

```
def printing_decorator(func):
    def wrapper(*args, **kwds):
        print("Calling", func)
        return func(*args, **kwds)
    return wrapper
```

We can use it to decorate function `add_forty_two`:

```
# A decorated function.
@printing_decorator
def add_forty_two(value: int) -> int:
    return value + 42

a = add_forty_two(3)
```

Since `printing_decorator` is not type-annotated, the following won't get type checked:

```
reveal_type(a)        # Revealed type is "Any"
add_forty_two('foo')  # No type checker error :(
```

This is a sorry state of affairs! If you run with `--strict`, mypy will even alert you to this fact: `Untyped decorator makes function "add_forty_two" untyped`

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.

Here's how one could annotate the decorator (Python 3.12 syntax):

```
from collections.abc import Callable
from typing import Any, cast

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

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

a = add_forty_two(3)
reveal_type(a)      # Revealed type is "builtins.int"
add_forty_two('x')  # Argument 1 to "add_forty_two" has incompatible type "str";
↪expected "int"
```

Here is the example using the legacy syntax (Python 3.11 and earlier):

```
from collections.abc import Callable
from typing import Any, TypeVar, cast

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

(continues on next page)

(continued from previous page)

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

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

a = add_forty_two(3)
reveal_type(a)      # Revealed type is "builtins.int"
add_forty_two('x')  # Argument 1 to "add_forty_two" has incompatible type "str";
↪expected "int"
```

This still has some shortcomings. First, we need to use the unsafe `cast()` to convince mypy that `wrapper()` has the same signature as `func` (see *casts*).

Second, the `wrapper()` function is not tightly type checked, although 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 `printing_decorator()`.

However, we can use a parameter specification, introduced using `**P`, for a more faithful type annotation (Python 3.12 syntax):

```
from collections.abc import Callable

def printing_decorator[**P, T](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
```

The same is possible using the legacy syntax with `ParamSpec` (Python 3.11 and earlier):

```
from collections.abc import Callable
from typing import TypeVar
from typing_extensions import ParamSpec

P = ParamSpec('P')
T = TypeVar('T')

def printing_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
```

Parameter specifications also allow you to describe decorators that alter the signature of the input function (Python 3.12 syntax):

```
from collections.abc import Callable
```

(continues on next page)

(continued from previous page)

```
# We reuse 'P' in the return type, but replace 'T' with 'str'
def stringify[**P, T](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)      # Revealed type is "builtins.str"
 add_forty_two('x')  # error: Argument 1 to "add_forty_two" has incompatible type "str";
↪expected "int"
```

Here is the above example using the legacy syntax (Python 3.11 and earlier):

```
from collections.abc import Callable
from typing import TypeVar
from typing_extensions import ParamSpec

P = ParamSpec('P')
T = TypeVar('T')

# 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
```

You can also insert an argument in a decorator (Python 3.12 syntax):

```
from collections.abc import Callable
from typing import Concatenate

def printing_decorator[**P, T](func: Callable[P, T]) -> Callable[Concatenate[str, P], T]:
    def wrapper(msg: str, /, *args: P.args, **kwds: P.kwargs) -> T:
        print("Calling", func, "with", msg)
        return func(*args, **kwds)
    return wrapper

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

a = add_forty_two('three', 3)
```

Here is the same function using the legacy syntax (Python 3.11 and earlier):

```
from collections.abc import Callable
from typing import TypeVar
from typing_extensions import Concatenate, ParamSpec
```

(continues on next page)

(continued from previous page)

```
P = ParamSpec('P')
T = TypeVar('T')

def printing_decorator(func: Callable[P, T]) -> Callable[Concatenate[str, P], T]:
    def wrapper(msg: str, /, *args: P.args, **kwds: P.kwargs) -> T:
        print("Calling", func, "with", msg)
        return func(*args, **kwds)
    return wrapper
```

Decorator factories

Functions that take arguments and return a decorator (also called second-order decorators), are similarly supported via generics (Python 3.12 syntax):

```
from collections.abc import Callable
from typing import Any

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

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

Note that mypy infers that `F` is used to make the `Callable` return value of `route` generic, instead of making `route` itself generic, since `F` is only used in the return type. Python has no explicit syntax to mark that `F` is only bound in the return value.

Here is the example using the legacy syntax (Python 3.11 and earlier):

```
from collections.abc import Callable
from typing import Any, 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` (Python 3.12 syntax):

```
from collections.abc import Callable
from typing import Any, overload

# Bare decorator usage
@overload
def atomic[F: Callable[..., Any]](func: F, /) -> F: ...
# Decorator with arguments
@overload
```

(continues on next page)

(continued from previous page)

```
def atomic[F: Callable[..., Any]](*, savepoint: bool = True) -> Callable[[F], F]: ...

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

# Usage
@atomic
def func1() -> None: ...

@atomic(savepoint=False)
def func2() -> None: ...
```

Here is the decorator from the example using the legacy syntax (Python 3.11 and earlier):

```
from collections.abc import Callable
from typing import Any, Optional, 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: Optional[Callable[..., Any]] = None, /, *, savepoint: bool = True):
    ...  # Same as above
```

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 3.12 syntax):

```
from typing import Protocol

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

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

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

(continues on next page)

(continued from previous page)

```
        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
```

Here is the definition of Box from the above example using the legacy syntax (Python 3.11 and earlier):

```
from typing import Protocol, TypeVar

T = TypeVar('T')

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

Note that `class ClassName(Protocol[T])` is allowed as a shorthand for `class ClassName(Protocol, Generic[T])` when using the legacy syntax, as per **PEP 544: Generic protocols**. This form is only valid when using the legacy syntax.

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

```
from typing import Protocol, TypeVar

T = TypeVar('T')

class ReadOnlyBox(Protocol[T]):  # error: Invariant type variable "T" used in protocol␣
↪where covariant one is expected
    def content(self) -> T: ...
```

This example correctly uses a covariant type variable:

```
from typing import Protocol, TypeVar

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 3.12 syntax):

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

class L:
    val: int
    def next(self) -> 'L': ...

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

result = last(L())
reveal_type(result)  # Revealed type is "builtins.int"
```

Here is the definition of Linked using the legacy syntax (Python 3.11 and earlier):

```
from typing import TypeVar

T = TypeVar('T')

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

1.14.11 Generic type aliases

Type aliases can be generic. In this case they can be used in two ways. First, subscripted aliases are equivalent to original types with substituted type variables. Second, unsubscripted aliases are treated as original types with type parameters replaced with Any.

The type statement introduced in Python 3.12 is used to define generic type aliases (it also supports non-generic type aliases):

```
from collections.abc import Callable, Iterable

type TInt[S] = tuple[int, S]
type UInt[S] = S | int
type CBack[S] = Callable[..., S]

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

type Vec[T: (int, float, complex)] = Iterable[tuple[T, T]]

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

def dilate[T: (int, float, complex)](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]]
```

(continues on next page)

(continued from previous page)

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

There is also a legacy syntax that relies on TypeVar. Here the number of type arguments must match the number of free type variables in the generic type alias definition. A type variables is free if it's not a type parameter of a surrounding class or function. Example (following **PEP 484: Type aliases**, Python 3.11 and earlier):

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

S = TypeVar('S')

TInt = tuple[int, S]  # 1 type parameter, since only S is free
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 (Python 3.12 syntax):

```
from example1 import AliasType
from example2 import Vec

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

def fun() -> AliasType:
    ...

type OIntVec = Vec[int] | None
```

Type aliases defined using the type statement are not valid as base classes, and they can't be used to construct instances:

```
from example1 import AliasType
from example2 import Vec
```

(continues on next page)

(continued from previous page)

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

class NewVec[T](Vec[T]):  # Error: not valid as base class
    ...

x = AliasType()  # Error: can't be used to create instances
```

Here are examples using the legacy syntax (Python 3.11 and earlier):

```
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:
    ...

OIntVec = Optional[Vec[int]]

T = TypeVar('T')

# Old-style type aliases can be used as base classes and you can
# construct instances using them

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

x = AliasType()

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

Using type variable bounds or value constraints in generic aliases has the same effect as in generic classes and functions.

1.14.12 Differences between the new and old syntax

There are a few notable differences between the new (Python 3.12 and later) and the old syntax for generic classes, functions and type aliases, beyond the obvious syntactic differences:

- Type variables defined using the old syntax create definitions at runtime in the surrounding namespace, whereas the type variables defined using the new syntax are only defined within the class, function or type variable that uses them.
- Type variable definitions can be shared when using the old syntax, but the new syntax doesn't support this.
- When using the new syntax, the variance of class type variables is always inferred.
- Type aliases defined using the new syntax can contain forward references and recursive references without using string literal escaping. The same is true for the bounds and constraints of type variables.
- The new syntax lets you define a generic alias where the definition doesn't contain a reference to a type parameter. This is occasionally useful, at least when conditionally defining type aliases.

- Type aliases defined using the new syntax can't be used as base classes and can't be used to construct instances, unlike aliases defined using the old syntax.

1.14.13 Generic class internals

You may wonder what happens at runtime when you index a generic class. Indexing returns a *generic alias* to the original class that returns instances of the original class on instantiation (Python 3.12 syntax):

```
>>> class Stack[T]: ...
>>> Stack
__main__.Stack
>>> Stack[int]
__main__.Stack[int]
>>> instance = Stack[int]()
>>> instance.__class__
__main__.Stack
```

Here is the same example using the legacy syntax (Python 3.11 and earlier):

```
>>> from typing import TypeVar, Generic
>>> T = TypeVar('T')
>>> class Stack(Generic[T]): ...
>>> Stack
__main__.Stack
>>> Stack[int]
__main__.Stack[int]
>>> instance = Stack[int]()
>>> instance.__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 the `Generic` base class that overloads the indexing operator using `__class_getitem__`. `typing.Generic` is included as an implicit base class even when using the new syntax:

```
>>> class Stack[T]: ...
>>> Stack.mro()
[<class '__main__.Stack'>, <class 'typing.Generic'>, <class 'object'>]
```

Note that in Python 3.8 and earlier, 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:

```
>>> # Prefer the 'list[int]' syntax
>>> from typing import List
>>> List[int]
typing.List[int]
```

Note that the generic aliases in `typing` don't support constructing instances, unlike the corresponding built-in classes:

```
>>> list[int]()
[]
>>> from typing import List
>>> List[int]()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: Type List cannot be instantiated; use list() instead
```

1.15 More types

This section introduces a few additional kinds of types, including NoReturn, NewType, 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.
- 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:

```
from typing import NoReturn

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

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

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

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

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

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:

```
class UserId(int):
    pass
```

(continues on next page)

(continued from previous page)

```
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:

```
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:

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

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

```
from typing import NewType

UserId = NewType('UserId', int)

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

UserId('user')          # Fails type check

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

num: 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 a *union type*, 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:

```
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:

```
UserId = int

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

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

But a similar example using NewType will not typecheck:

```
from typing import NewType

UserId = NewType('UserId', int)

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

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

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 types*. 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:

```
def mouse_event(x1: int,
                y1: int,
                x2: int | None = None,
                y2: int | None = None) -> 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 overload

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

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

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

def mouse_event(x1: int,
                y1: int,
                x2: int | None = None,
                y2: int | None = None) -> 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 3.12 syntax):

```python
from collections.abc import Sequence
from typing import overload

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

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

    def __getitem__(self, index: int | slice) -> T | Sequence[T]:
```

(continues on next page)

(continued from previous page)

```
        if isinstance(index, int):
            # Return a T here
        elif isinstance(index, slice):
            # Return a sequence of Ts here
        else:
            raise TypeError(...)
```

Here is the same example using the legacy syntax (Python 3.11 and earlier):

```
from collections.abc import Sequence
from typing import TypeVar, 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: int | slice) -> T | Sequence[T]:
        if isinstance(index, int):
            # Return a T here
        elif isinstance(index, slice):
            # Return a sequence of Ts here
        else:
            raise TypeError(...)
```

Note

If you just need to constrain a type variable to certain types or subtypes, you can use a *value-constrained type variable*.

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:

```
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:

```
# 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 implementer is responsible for making sure `summarize` breaks ties in the same way at runtime.

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

```
dynamic_var: Any = some_dynamic_function()

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

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

```
some_list: list[int] | list[str]

# output3 is of type '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:

```
from typing import overload

class Expression:
    # ...snip...

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

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

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

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

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

While this code snippet is technically type-safe, it does contain an anti-pattern: the second variant will never be selected!

If we try calling `add(Literal(3), Literal(4))`, mypy will always pick the first variant and evaluate the function call to be of type `Expression`, not `Literal`. This is because `Literal` is a subtype of `Expression`, which means the "pick the first match" rule will always halt after considering the first overload.

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

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

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

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

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

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

```
from typing import overload

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

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

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

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

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

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

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

1. All of the arguments of the first variant are potentially 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.

Note that in cases where you ignore the overlapping overload error, mypy will usually still infer the types you expect at callsites.

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

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

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

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

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

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: int | slice` and a return type of `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 `int | slice` and `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 `int | slice` and `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.

```
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
```

(continues on next page)

(continued from previous page)

```
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.

```
from typing import Any, overload

class A: ...
class B: ...


def g(bool_var: bool) -> None:
    if bool_var:  # Condition can't be inferred, unable to merge overloads
        @overload
        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 3.12 syntax):

```
class Tag[T]:
    item: T

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

(continues on next page)

(continued from previous page)

```
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 3.12 syntax):

```
from collections.abc import Sequence

class Storage[T]:
    def __init__(self, content: T) -> None:
        self._content = content

    def first_chunk[S](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␣
↪function
                          # "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 3.12 syntax):

```
from collections.abc import Callable
from typing import overload

class Tag[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: ...
```

(continues on next page)

(continued from previous page)

```
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 *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 (Python 3.12 syntax):

```
from typing import Self

class Base[T]:
    def __init__(self, item: T) -> None:
        self.item = item

    @classmethod
    def make_pair(cls, item: T) -> tuple[Self, Self]:
        return cls(item), cls(item)

class Sub[T](Base[T]):
    ...

pair = Sub.make_pair('yes')  # Type is "tuple[Sub[str], Sub[str]]"
bad = Sub[int].make_pair('no')  # Error: Argument 1 to "make_pair" of "Base"
                                # has incompatible type "str"; expected "int"
```

1.15.5 Typing async/await

Mypy lets you type coroutines that use the `async/await` syntax. For more information regarding coroutines, see **PEP 492** and the asyncio documentation.

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

```
import asyncio

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

async def countdown(tag: str, count: int) -> str:
    while count > 0:
        my_str = await format_string(tag, count)  # type is inferred to be str
        print(my_str)
        await asyncio.sleep(0.1)
        count -= 1
    return "Blastoff!"

asyncio.run(countdown("Millennium Falcon", 5))
```

The result of calling an `async def` function *without awaiting* will automatically be inferred to be a value of type `Coroutine[Any, Any, T]`, which is a subtype of `Awaitable[T]`:

```
my_coroutine = countdown("Millennium Falcon", 5)
reveal_type(my_coroutine)  # Revealed type is "typing.Coroutine[Any, Any, builtins.str]"
```

Asynchronous iterators

If you have an asynchronous iterator, you can use the `AsyncIterator` type in your annotations:

```
from collections.abc import AsyncIterator
from typing import Optional
import asyncio

class arange:
    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 run_countdown(tag: str, countdown: AsyncIterator[int]) -> str:
    async for i in countdown:
        print(f'T-minus {i} ({tag})')
        await asyncio.sleep(0.1)
    return "Blastoff!"
```

(continues on next page)

(continued from previous page)

```
asyncio.run(run_countdown("Serenity", arange(5, 0, -1)))
```

Async generators (introduced in **PEP 525**) are an easy way to create async iterators:

```
from collections.abc import AsyncGenerator
from typing import Optional
import asyncio

# Could also type this as returning AsyncIterator[int]
async def arange(start: int, stop: int, step: int) -> AsyncGenerator[int, None]:
    current = start
    while (step > 0 and current < stop) or (step < 0 and current > stop):
        yield current
        current += step

asyncio.run(run_countdown("Battlestar Galactica", arange(5, 0, -1)))
```

One common confusion is that the presence of a `yield` statement in an `async def` function has an effect on the type of the function:

```
from collections.abc import AsyncIterator

async def arange(stop: int) -> AsyncIterator[int]:
    # When called, arange gives you an async iterator
    # Equivalent to Callable[[int], AsyncIterator[int]]
    i = 0
    while i < stop:
        yield i
        i += 1

async def coroutine(stop: int) -> AsyncIterator[int]:
    # When called, coroutine gives you something you can await to get an async iterator
    # Equivalent to Callable[[int], Coroutine[Any, Any, AsyncIterator[int]]]
    return arange(stop)

async def main() -> None:
    reveal_type(arange(5))  # Revealed type is "typing.AsyncIterator[builtins.int]"
    reveal_type(coroutine(5))  # Revealed type is "typing.Coroutine[Any, Any, typing.
↪AsyncIterator[builtins.int]]"

    await arange(5)  # Error: Incompatible types in "await" (actual type
↪"AsyncIterator[int]", expected type "Awaitable[Any]")
    reveal_type(await coroutine(5))  # Revealed type is "typing.AsyncIterator[builtins.
↪int]"
```

This can sometimes come up when trying to define base classes, Protocols or overloads:

```
from collections.abc import AsyncIterator
from typing import Protocol, overload

class LauncherIncorrect(Protocol):
    # Because launch does not have yield, this has type
```

(continues on next page)

(continued from previous page)

```
    # Callable[[], Coroutine[Any, Any, AsyncIterator[int]]]
    # instead of
    # Callable[[], AsyncIterator[int]]
    async def launch(self) -> AsyncIterator[int]:
        raise NotImplementedError

class LauncherCorrect(Protocol):
    def launch(self) -> AsyncIterator[int]:
        raise NotImplementedError

class LauncherAlsoCorrect(Protocol):
    async def launch(self) -> AsyncIterator[int]:
        raise NotImplementedError
        if False:
            yield 0

# The type of the overloads is independent of the implementation.
# In particular, their type is not affected by whether or not the
# implementation contains a `yield`.
# Use of `def`` makes it clear the type is Callable[..., AsyncIterator[int]],
# whereas with `async def` it would be Callable[..., Coroutine[Any, Any,␣
↪AsyncIterator[int]]]
@overload
def launch(*, count: int = ...) -> AsyncIterator[int]: ...
@overload
def launch(*, time: float = ...) -> AsyncIterator[int]: ...

async def launch(*, count: int = 0, time: float = 0) -> AsyncIterator[int]:
    # The implementation of launch is an async generator and contains a yield
    yield 0
```

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:

```
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: ...
```

(continues on next page)

(continued from previous page)

```
# 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` package.

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]`, `Literal[4] | None`, and `Optional[Literal[4]]` are all equivalent.

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

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

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

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

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

Declaring literal variables

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

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

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

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

If you find repeating the value of the variable in the type hint to be tedious, you can instead change the variable to be `Final` (see *Final names, methods and classes*):

```
from typing import Final, Literal

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

c: Final = 19

reveal_type(c)          # Revealed type is "Literal[19]?"
expects_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:

```
from typing import Final, Literal

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

reveal_type(19)
expects_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:

```
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]].
```

(continues on next page)

(continued from previous page)

```
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:

```
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
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"

# You can also index using unions of literals
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 TypedDicts 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:

```
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.
    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 3.12 syntax):

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

def process(w: 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:

```
from typing import Literal

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

def validate(x: PossibleValues) -> bool:
    if x == 'one':
        return True
    elif x == 'two':
        return False
    raise ValueError(f'Invalid value: {x}')

assert validate('one') is True
assert validate('two') is False
```

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
from typing_extensions import assert_never

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

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):

```
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

Mypy has special support for `enum.Enum` and its subclasses: `enum.IntEnum`, `enum.Flag`, `enum.IntFlag`, and `enum.StrEnum`.

```
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:

```
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:

```
from enum import Enum
from typing import NoReturn
from typing_extensions import assert_never

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

Now, let's use an exhaustiveness check:

```
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:

```
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). For match statements specifically, inexhaustive matches can be caught without needing to use `assert_never` by using *--enable-error-code exhaustive-match*.

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:

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

  Mypy also catches this error:

```
class AllDirection(Direction):  # E: Cannot inherit from final class "Direction"
    left = 'left'
    right = 'right'
```

- All Enum fields are implicitly final as well.

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

- All field names are checked to be unique.

```
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.

```
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 TypedDict

Python programs often use dictionaries with string keys to represent objects. TypedDict lets you give precise types for dictionaries that represent objects with a fixed schema, such as {'id': 1, 'items': ['x']}.

Here is a typical example:

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

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

```
from typing 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:

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

Mypy will detect an invalid key as an error:

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

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

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

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

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

Like all types, `TypedDict`s can be used as components to build arbitrarily complex types. For example, you can define nested `TypedDict`s and containers with `TypedDict` items. Unlike most other types, mypy uses structural compatibility checking (or structural subtyping) with `TypedDict`s. 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:

```
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`:

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

1.17.1 Totality

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

```
# Error: 'year' missing
toy_story: 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:

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

(continues on next page)

(continued from previous page)

```
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:

```
# 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.

1.17.2 Supported operations

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

- Anything included in 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.

1.17.3 Class-based syntax

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

```
from typing import TypedDict  # "from typing_extensions" in Python 3.7 and earlier

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:

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

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

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

1.17.4 Mixing required and non-required items

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

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

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

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

1.17.5 Read-only items

You can use `typing.ReadOnly`, introduced in Python 3.13, or `typing_extensions.ReadOnly` to mark TypedDict items as read-only (**PEP 705**):

```
from typing import TypedDict

# Or "from typing ..." on Python 3.13+
from typing_extensions import ReadOnly

class Movie(TypedDict):
    name: ReadOnly[str]
    num_watched: int
```

(continues on next page)

(continued from previous page)

```
m: Movie = {"name": "Jaws", "num_watched": 1}
m["name"] = "The Godfather"  # Error: "name" is read-only
m["num_watched"] += 1  # OK
```

A TypedDict with a mutable item can be assigned to a TypedDict with a corresponding read-only item, and the type of the item can vary *covariantly*:

```
class Entry(TypedDict):
    name: ReadOnly[str | None]
    year: ReadOnly[int]

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

def process_entry(i: Entry) -> None: ...

m: Movie = {"name": "Jaws", "year": 1975}
process_entry(m)  # OK
```

1.17.6 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.

1.17.7 Inline TypedDict types

Note

This is an experimental (non-standard) feature. Use `--enable-incomplete-feature=InlineTypedDict` to enable.

Sometimes you may want to define a complex nested JSON schema, or annotate a one-off function that returns a TypedDict. In such cases it may be convenient to use inline TypedDict syntax. For example:

```
def test_values() -> {"width": int, "description": str}:
    return {"width": 42, "description": "test"}

class Response(TypedDict):
    status: int
    msg: str
    # Using inline syntax here avoids defining two additional TypedDicts.
    content: {"items": list[{"key": str, "value": str}]}
```

Inline TypedDicts can also by used as targets of type aliases, but due to ambiguity with a regular variables it is only allowed for (newer) explicit type alias forms:

```
from typing import TypeAlias

X = {"a": int, "b": int}  # creates a variable with type dict[str, type[int]]
Y: TypeAlias = {"a": int, "b": int}  # creates a type alias
type Z = {"a": int, "b": int}  # same as above (Python 3.12+ only)
```

Also, due to incompatibility with runtime type-checking it is strongly recommended to *not* use inline syntax in union types.

1.18 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.

1.18.1 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 to prevent unintended modification. Mypy will prevent further assignments to final names in type-checked code:

```
from typing import Final

RATE: Final = 3_000

class Base:
    DEFAULT_ID: Final = 0

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

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

```
from typing 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:

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

  Here, mypy will infer type int for ID.

- You can omit the type:

  ```
  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 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):

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

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

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

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

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

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

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

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

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

Declaring a name as final only guarantees that the name 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:

```
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
```

1.18.2 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:

```
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):

```
from typing import final, overload

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

1.18.3 Final classes

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

```
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.

```
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
```

1.19 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.

1.19.1 Defining a metaclass

```
class M(type):
    pass

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

1.19.2 Metaclass usage example

Mypy supports the lookup of attributes in the metaclass:

```
from typing import ClassVar, TypeVar

S = TypeVar("S")

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

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

class A(metaclass=M):
    pass

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

class B(A):
    pass

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

1.19.3 Gotchas and limitations of metaclass support

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

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

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

class B1(A1, metaclass=M2): pass  # Mypy Error: metaclass conflict
# 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

class B12(A1, A2): pass  # Mypy Error: metaclass conflict

# This can be solved via a common metaclass subtype:
class CorrectMeta(M1, M2): pass
class B2(A1, A2, metaclass=CorrectMeta): pass  # OK, runtime is also OK
```

- 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.

- Self is not allowed as annotation in metaclasses as per PEP 673.

For some builtin types, mypy may think their metaclass is abc.ABCMeta even if it is type at runtime. In those cases, you can either:

- use abc.ABCMeta instead of type as the superclass of your metaclass if that works in your use-case
- mute the error with # type: ignore[metaclass]

```
import abc

assert type(tuple) is type  # metaclass of tuple is type at runtime

# The problem:
class M0(type): pass
class A0(tuple, metaclass=M0): pass  # Mypy Error: metaclass conflict

# Option 1: use ABCMeta instead of type
class M1(abc.ABCMeta): pass
class A1(tuple, metaclass=M1): pass

# Option 2: mute the error
class M2(type): pass
class A2(tuple, metaclass=M2): pass  # type: ignore[metaclass]
```

1.20 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.

1.20.1 Specifying code to be checked

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

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

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

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

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

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

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

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

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

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

```
$ mypy -p html
```

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

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

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

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

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

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

You can also use the `files` option in your `mypy.ini` file to specify which files to check, in which case you can simply run `mypy` with no arguments.

1.20.2 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.

1.20.3 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:

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

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

- 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 `--exclude`.
- For each file to be checked, mypy will attempt to associate the file (e.g. `project/foo/bar/baz.py`) with a fully qualified module name (e.g. `foo.bar.baz`). The directory the package is in (`project`) is then added to mypy's module search paths.

How mypy determines fully qualified module names depends on if the options `--no-namespace-packages` and `--explicit-package-bases` are set.

1. If `--no-namespace-packages` is set, mypy will rely solely upon the presence of `__init__.py[i]` 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. The default case. If `--namespace-packages` is on, but `--explicit-package-bases` is off, mypy will allow for the possibility that directories without `__init__.py[i]` are packages. Specifically, mypy will look at all parent directories of the file and use the location of the highest `__init__.py[i]` 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]`, the module name assumed is `__main__` (matching the behavior of the Python interpreter), unless `--scripts-are-modules` is passed.

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

1.20.4 How mypy handles imports

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

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

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

1.20.5 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"
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:

```
# 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()
```

This can result in mypy failing to warn you about errors in your code. Since operations on `Any` result in `Any`, these dynamic types can propagate through your code, making type checking less effective. See *Dynamically typed code* for more information.

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 to obtain type hints for the library you're using:

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*.
4. Force mypy to analyze the library as best as it can (as if the library provided a `py.typed` file), despite it likely missing any type annotations. In general, the quality of type checking will be poor and mypy may have issues when analyzing code not designed to be type checked.

   You can do this via setting the `--follow-untyped-imports` command line flag or `follow_untyped_imports` config file option to True. This option can be specified on a per-module basis as well:

```
[mypy-untyped_package.*]
follow_untyped_imports = True
```

```
[[tool.mypy.overrides]]
module = ["untyped_package.*"]
follow_untyped_imports = true
```

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`, and mypy may not catch errors in its use.

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 per-module section to your *mypy config file* setting *ignore_missing_imports* to True for that library. 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
   ```

   ```
   [[tool.mypy.overrides]]
   module = ["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* untyped libraries in your codebase, use *--disable-error-code=import-untyped*. See *Check that import target can be found [import-untyped]* for more details on this error code.

   You can also set *disable_error_code*, like so:

   ```
   [mypy]
   disable_error_code = import-untyped
   ```

   ```
   [tool.mypy]
   disable_error_code = ["import-untyped"]
   ```

   You can also set 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. We recommend avoiding `--ignore-missing-imports` if possible: 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"
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 commands. If you're running mypy in CI, you can ensure the presence of any stub packages you need the same as you would any other test dependency, e.g. by adding them to the appropriate `requirements.txt` file.

Alternatively, add the *--install-types* to your mypy command to install all known missing stubs:

```
mypy --install-types
```

This is slower than explicitly installing stubs, since it effectively runs mypy 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. It also can make controlling stub versions harder, resulting in less reproducible type checking.

By default, *--install-types* shows a confirmation prompt. Use *--non-interactive* to install all suggested stub packages without asking for confirmation *and* type check your code:

If you've already installed the relevant third-party libraries in an environment other than the one mypy is running in, you can use *--python-executable* flag to point to the Python executable for that environment, and mypy will find packages installed for that Python executable.

If you've installed the relevant stub packages and are still getting this error, see the *section below*.

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.

   You can confirm that you are running mypy from the environment you expect by running it like `python -m mypy ...`. You can confirm that you are installing into the environment you expect by running pip like `python -m pip ...`.
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`).

1.20.6 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 list of directories, colon-separated on UNIX systems, semicolon-separated on Windows).

- 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.)

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.

1.20.7 Following imports

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

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

How do we want mypy to type check `mycode.bar`? Mypy's behaviour here is configurable – although we **strongly recommend** using the default – by using the *--follow-imports* flag. This flag accepts one of four string values:

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

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

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

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

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

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

> **Warning**
>
> We do not recommend using `follow_imports=skip` unless you're really sure you know what you are doing. This option greatly restricts the analysis mypy can perform and you will lose a lot of the benefits of type checking.
>
> This is especially true at the global level. Setting a per-module `follow_imports=skip` for a specific problematic module can be useful without causing too much harm.

> **Note**
>
> If you're looking to resolve import errors related to libraries, try following the advice in *Missing imports* before messing with `follow_imports`.

1.21 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.

1.21.1 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 discovery of files via *import following*. You can use a per-module `ignore_errors` config option to silence errors from a given module, or 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`.

--exclude-gitignore

This flag will add everything that matches `.gitignore` file(s) to `--exclude`.

1.21.2 Optional arguments

-h, --help

Show help message and exit.

-v, --verbose

More verbose messages.

-V, --version

Show program's version number and exit.

-O FORMAT, **--output** FORMAT {json}

Set a custom output format.

1.21.3 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`.)

1.21.4 Import discovery

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

--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 the absence of *__init__.py*. 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:

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

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

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

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

For more details, see *Missing imports*.

--follow-untyped-imports

This flag makes mypy analyze imports from installed packages even if missing a *py.typed marker or stubs*.

> **Warning**
>
> Note that analyzing all unannotated modules might result in issues when analyzing code not designed to be type checked and may significantly increase how long mypy takes to run.

--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.

--no-namespace-packages

This flag disables import discovery of namespace packages (see **PEP 420**). In particular, this prevents discovery of packages that don't have an `__init__.py` (or `__init__.pyi`) file.

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.

1.21.5 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.

1.21.6 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.

1.21.7 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.

--untyped-calls-exclude

This flag allows one to selectively disable *--disallow-untyped-calls* for functions and methods defined in specific packages, modules, or classes. Note that each exclude entry acts as a prefix. For example (assuming there are no type annotations for `third_party_lib` available):

```
# mypy --disallow-untyped-calls
#      --untyped-calls-exclude=third_party_lib.module_a
#      --untyped-calls-exclude=foo.A
from third_party_lib.module_a import some_func
from third_party_lib.module_b import other_func
import foo

some_func()  # OK, function comes from module `third_party_lib.module_a`
other_func()  # E: Call to untyped function "other_func" in typed context

foo.A().meth()  # OK, method was defined in class `foo.A`
foo.B().meth()  # E: Call to untyped function "meth" in typed context

# file foo.py
class A:
    def meth(self): pass
class B:
    def meth(self): pass
```

--disallow-untyped-defs

This flag reports an error whenever it encounters a function definition without type annotations or with incomplete type annotations. (a superset of *--disallow-incomplete-defs*).

For example, it would report an error for `def f(a, b)` and `def f(a: int, b)`.

--disallow-incomplete-defs

This flag reports an error whenever it encounters a partly annotated function definition, while still allowing entirely unannotated definitions.

For example, it would report an error for `def f(a: int, b)` but not `def f(a, b)`.

--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.

1.21.8 None and Optional handling

The following flags adjust how mypy handles values of type `None`.

`--implicit-optional`

This flag causes mypy to treat parameters with a `None` default value as having an implicit optional type (`T | None`).

For example, if this flag is set, mypy would assume that the `x` parameter is actually of type `int | None` in the code snippet below, since the default parameter is `None`:

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

Note: This was disabled by default starting in mypy 0.980.

`--no-strict-optional`

This flag effectively disables checking of optional types and `None` values. With this option, mypy doesn't generally check the use of `None` values – it is treated as compatible with every type.

> **Warning**
>
> `--no-strict-optional` is evil. Avoid using it and definitely do not use it without understanding what it does.

1.21.9 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 and is marked as an abstract method, is in a protocol class, or is in a stub file
- **The execution path can never return; for example, if an exception**
  is always raised

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.

```
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 three cases:

1. When the unreachable statement is a `raise` statement, is 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 is `return NotImplemented`. This is allowed by mypy due to its use in operator overloading.
3. 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 *value-constrained type variables*.

This limitation will be removed in future releases of mypy.

--report-deprecated-as-note

If error code `deprecated` is enabled, mypy emits errors if your code imports or uses deprecated features. This flag converts such errors to notes, causing mypy to eventually finish with a zero exit code. Features are considered deprecated when decorated with `warnings.deprecated`.

--deprecated-calls-exclude

This flag allows one to selectively disable *deprecated* warnings for functions and methods defined in specific packages, modules, or classes. Note that each exclude entry acts as a prefix. For example (assuming `foo.A.func` is deprecated):

```
# mypy --enable-error-code deprecated
#      --deprecated-calls-exclude=foo.A
import foo

foo.A().func()  # OK, the deprecated warning is ignored

# file foo.py
from typing_extensions import deprecated
class A:
```

(continues on next page)

(continued from previous page)

```
    @deprecated("Use A.func2 instead")
    def func(self): pass
```

1.21.10 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 the redefinition of *unannotated* variables with an arbitrary type. This also requires `--local-partial-types`, which is enabled by default starting from mypy 2.0. Example:

```
def maybe_convert(n: int, b: bool) -> int | str:
    if b:
        x = str(n)  # Assign "str"
    else:
        x = n       # Assign "int"
    # Type of "x" is "int | str" here.
    return x
```

Without this flag, mypy only supports inferring optional types (`X | None`) from multiple assignments. With this option enabled, mypy can infer arbitrary union types.

This also enables an unannotated variable to have different types in different code locations:

```
if check():
    for x in range(n):
        # Type of "x" is "int" here.
        ...
else:
    for x in ['a', 'b']:
        # Type of "x" is "str" here.
        ...
```

Function arguments are special, changing their type within function body is allowed even if they are annotated, but that annotation is used to infer types of r.h.s. of all subsequent assignments. Such middle-ground semantics provides good balance for majority of common use cases. For example:

```
def process(values: list[float]) -> None:
    if not values:
        values = [0, 0, 0]
    reveal_type(values)  # Revealed type is list[float]
```

Note: We are planning to turn this flag on by default in a future mypy release.

--allow-redefinition-new

Deprecated alias for `--allow-redefinition`.

--allow-redefinition-old

This is an older, more limited variant of `--allow-redefinition`. 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.

We have no plans to remove this flag, but `--allow-redefinition` is recommended for new use cases.

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
```

--no-local-partial-types

Disable local partial types to enable legacy type inference mode for containers.

Local partial types prevent inferring a container type for a variable, when the initial assignment happens at module top level or in a class body, and the container item type is only set in a function. Example:

```
a = []  # Need type annotation unless using --no-local-partial-types

def func() -> None:
    a.append(1)

reveal_type(a)  # "list[int]" if using --no-local-partial-types
```

Local partial types are enabled by default starting from mypy 2.0. The mypy daemon requires local partial types.

--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:

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

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

assert text is not None  # OK, check against None is allowed
```

--strict-equality-for-none

This flag extends *--strict-equality* for checks against `None`:

```
text: str
assert text is not None  # Error: non-overlapping identity check!
```

Note that *--strict-equality-for-none* only works in combination with *--strict-equality*.

--no-strict-bytes

Treat `bytearray` and `memoryview` as subtypes of `bytes`. This is not true at runtime and can lead to unexpected behavior. This was the default behavior prior to mypy 2.0.

```
def f(buf: bytes) -> None:
    assert isinstance(buf, bytes)  # Raises runtime AssertionError with bytearray/
↪memoryview
    with open("binary_file", "wb") as fp:
        fp.write(buf)

# Using --no-strict-bytes disables the following errors
f(bytearray(b""))  # Argument 1 to "f" has incompatible type "bytearray"; expected
↪"bytes"
f(memoryview(b""))  # Argument 1 to "f" has incompatible type "memoryview";
↪expected "bytes"

# If `f` accepts any object that implements the buffer protocol,
# consider using Buffer instead:
from collections.abc import Buffer  # "from typing_extensions" in Python 3.11 and
↪earlier

def f(buf: Buffer) -> None:
    with open("binary_file", "wb") as fp:
        fp.write(buf)

f(b"")  # Ok
f(bytearray(b""))  # Ok
f(memoryview(b""))  # Ok
```

--extra-checks

This flag enables additional checks that are technically correct but may be impractical. In particular, it prohibits partial overlap in `TypedDict` updates, and makes arguments prepended via `Concatenate` positional-only. For example:

```python
from typing import TypedDict

class Foo(TypedDict):
    a: int

class Bar(TypedDict):
    a: int
    b: int

def test(foo: Foo, bar: Bar) -> None:
    # This is technically unsafe since foo can have a subtype of Foo at
    # runtime, where type of key "b" is incompatible with int, see below
    bar.update(foo)

class Bad(Foo):
    b: str
bad: Bad = {"a": 0, "b": "no"}
test(bad, bar)
```

In future more checks may be added to this flag if:

- The corresponding use cases are rare, thus not justifying a dedicated strictness flag.
- The new check cannot be supported as an opt-in error code.

--strict

This flag mode enables a defined subset of optional error-checking flags. This subset primarily includes checks for inadvertent type unsoundness (i.e strict will catch type errors as long as intentional methods like type ignore or casting were not used.)

Note: the `--warn-unreachable` flag is not automatically enabled by the strict flag.

The strict flag does not take precedence over other strict-related flags. Directly specifying a flag of alternate behavior will override the behavior of strict, regardless of the order in which they are passed. 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'
```

(continues on next page)

(continued from previous page)

```
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]
```

1.21.11 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:

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

Mypy normally displays an error message that looks like this:

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

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

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

--show-column-numbers

This flag will add column offsets to error messages. For example, the following indicates an error in line 12, column 9 (note that column offsets are 0-based):

```
main.py:12:9: error: Unsupported operand types for / ("int" and "str")
```

--show-error-code-links

This flag will also display a link to error code documentation, anchored to the error code reported by mypy. The corresponding error code will be highlighted within the documentation page. If we enable this flag, the error message now looks like this:

```
main.py:3: error: Unsupported operand types for - ("int" and "str")  [operator]
main.py:3: note: See 'https://mypy.rtfd.io/en/stable/_refs.html#code-operator' for␣
↪more info
```

--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`.

--hide-error-codes

This flag will hide the error code `[<code>]` from error messages. By default, 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 -1.

1.21.12 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).

--no-sqlite-cache

Avoid using SQLite database to store the cache, instead write cache data out to individual files.

--cache-fine-grained

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

--skip-version-check

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

--skip-cache-mtime-checks

Skip cache internal consistency checks based on mtime.

1.21.13 Parallel type-checking

By default, mypy checks all modules in the same Python process. This can be slow for large code bases. Mypy offers experimental parallel type-checking mode using multiple worker processes. In parallel mode, modules that do not depend om each other are type-checked in parallel. *Incremental cache* is used to manage most of the shared state. Parallel type-checking also requires `--local-partial-types`, which is enabled by default starting from mypy 2.0.

-n NUMBER, **--num-workers** NUMBER

Use `NUMBER` parallel worker processes (in addition to the coordinator process) to perform type-checking. Specifying `--num-workers 0` (default) disables parallel checking. Automatic detection of the optimal number of workers is not supported yet.

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

Notes:

- An import cycle is always processed as a whole by a worker process. Thus, avoiding large import cycles in your code will *significantly* improve type-checking speed.
- Specifying a number of workers that is larger than the number of *physical* CPU cores is not beneficial, since mypy is usually CPU bound. Best way to tune the number of workers on a given machine is to start from 3-4 workers and increase the number while you see a performance improvement.
- Parallel mode requires and automatically enables `--native-parser`.

1.21.14 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. To test third-party stubs, for example try `MYPYPATH=stubs/six mypy ...`.

--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.

--native-parser

This enables fast Rust-based parser that parses directly to mypy AST. It will become the default parser in one of the next mypy releases.

1.21.15 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].

1.21.16 Enabling incomplete/experimental features

--enable-incomplete-feature {PreciseTupleTypes,InlineTypedDict,TypeForm}

Some features may require several mypy releases to implement, for example due to their complexity, potential for backwards incompatibility, or ambiguous semantics that would benefit from feedback from the community. You can enable such features for early preview using this flag. Note that it is not guaranteed that all features will be ultimately enabled by default. In *rare cases* we may decide to not go ahead with certain features.

List of currently incomplete/experimental features:

- `PreciseTupleTypes`: this feature will infer more precise tuple types in various scenarios. Before variadic types were added to the Python type system by **PEP 646**, it was impossible to express a type like "a tuple with at least two integers". The best type available was `tuple[int, ...]`. Therefore, mypy applied very lenient checking for variable-length tuples. Now this type can be expressed as `tuple[int, int, *tuple[int, ...]]`. For such more precise types (when explicitly *defined* by a user) mypy, for example, warns about unsafe index access, and generally handles them in a type-safe manner. However, to avoid problems in existing code, mypy does not *infer* these precise types when it technically can. Here are notable examples where `PreciseTupleTypes` infers more precise types:

```
numbers: tuple[int, ...]

more_numbers = (1, *numbers, 1)
reveal_type(more_numbers)
# Without PreciseTupleTypes: tuple[int, ...]
# With PreciseTupleTypes: tuple[int, *tuple[int, ...], int]

other_numbers = (1, 1) + numbers
reveal_type(other_numbers)
# Without PreciseTupleTypes: tuple[int, ...]
# With PreciseTupleTypes: tuple[int, int, *tuple[int, ...]]

if len(numbers) > 2:
    reveal_type(numbers)
    # Without PreciseTupleTypes: tuple[int, ...]
    # With PreciseTupleTypes: tuple[int, int, int, *tuple[int, ...]]
else:
    reveal_type(numbers)
    # Without PreciseTupleTypes: tuple[int, ...]
    # With PreciseTupleTypes: tuple[()] | tuple[int] | tuple[int, int]
```

- `InlineTypedDict`: this feature enables non-standard syntax for inline *TypedDicts*, for example:

```
def test_values() -> {"width": int, "description": str}:
    return {"width": 42, "description": "test"}
```

- `TypeForm`: this feature enables `TypeForm`, as described in *PEP 747 – Annotating Type Forms <https://peps.python.org/pep-0747/>_*.

1.21.17 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_OUTPUT_FILE`

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.

--junit-format `{global,per_file}`

If –junit-xml is set, specifies format. global (default): single test with all errors; per_file: one test entry per file with failures.

--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.)

1.22 The mypy configuration file

Mypy is very configurable. This is most useful when introducing typing to an existing codebase. See *Using mypy with an existing codebase* for concrete advice for that situation.

Mypy supports reading configuration settings from a file. By default, mypy will discover configuration files by walking up the file system (up until the root of a repository or the root of the filesystem). In each directory, it will look for the following configuration files (in this order):

1. `mypy.ini`
2. `.mypy.ini`
3. `pyproject.toml` (containing a `[tool.mypy]` section)
4. `setup.cfg` (containing a `[mypy]` section)

If no configuration file is found by this method, mypy will then look for configuration files in the following locations (in this order):

1. `$XDG_CONFIG_HOME/mypy/config`
2. `~/.config/mypy/config`
3. `~/.mypy.ini`

The `--config-file` command-line flag has the highest precedence and must point towards a valid configuration file; otherwise mypy will report an error and exit. Without the command line option, mypy will look for configuration files in the precedence order above.

It is important to understand that there is no merging of configuration files, as it would lead to ambiguity.

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}`.

1.22.1 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.

1.22.2 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]).

1.22.3 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).

1.22.4 Example mypy.ini

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

```
# Global options:

[mypy]
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.

2. Selectively *disable* the "function is returning any" warnings within `mycode.bar` only. This overrides the global default we set earlier.
3. Suppress any error messages generated when your codebase tries importing the module `somelibrary`. This is useful if `somelibrary` is some 3rd party library missing type hints.

1.22.5 Import discovery

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

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]`).

modules

Type
comma-separated list of strings

A comma-separated list of packages which should be checked by mypy if none are given on the command line. Mypy *will not* recursively type check any submodules of the provided module.

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

packages

Type
comma-separated list of strings

A comma-separated list of packages which should be checked by mypy if none are given on the command line. Mypy *will* recursively type check any submodules of the provided package. This flag is identical to `modules` apart from this behavior.

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.

```
[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:

```
[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:

```
[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*.

`exclude_gitignore`

Type
boolean

Default
False

This flag will add everything that matches `.gitignore` file(s) to *exclude*. This option may only be set in the global section (`[mypy]`).

namespace_packages

Type
boolean

Default
True

Enables **PEP 420** style namespace packages. See the corresponding flag *--no-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 the absence of *__init__.py*. 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_untyped_imports

Type
boolean

Default
False

Makes mypy analyze imports from installed packages even if missing a *py.typed marker or stubs*.

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.

> **Warning**
>
> Note that analyzing all unannotated modules might result in issues when analyzing code not designed to be type checked and may significantly increase how long mypy takes to run.

`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 *PEP 561 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
boolean

Default
False

Disables using type information in installed packages (see **PEP 561**). 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 **PEP 561** 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]`).

1.22.6 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 `3.10`. 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.

1.22.7 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

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

`disallow_any_generics`

Type
boolean

Default
False

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

`disallow_subclassing_any`

Type
boolean

Default
False

Disallows subclassing a value of type `Any`.

1.22.8 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

Disallows calling functions without type annotations from functions with type annotations. Note that when used in per-module options, it enables/disables this check **inside** the module(s) specified, not for functions that come from that module(s), for example config like this:

```
[mypy]
disallow_untyped_calls = True

[mypy-some.library.*]
disallow_untyped_calls = False
```

will disable this check inside `some.library`, not for your code that imports `some.library`. If you want to selectively disable this check for all your code that imports `some.library` you should instead use *untyped_calls_exclude*, for example:

```
[mypy]
disallow_untyped_calls = True
untyped_calls_exclude = some.library
```

untyped_calls_exclude

Type
comma-separated list of strings

Selectively excludes functions and methods defined in specific packages, modules, and classes from action of *disallow_untyped_calls*. This also applies to all submodules of packages (i.e. everything inside a given prefix). Note, this option does not support per-file configuration, the exclusions list is defined globally for all your code.

disallow_untyped_defs

Type
boolean

Default
False

Disallows defining functions without type annotations or with incomplete type annotations (a superset of *disallow_incomplete_defs*).

For example, it would report an error for `def f(a, b)` and `def f(a: int, b)`.

disallow_incomplete_defs

Type
boolean

Default
False

Disallows defining functions with incomplete type annotations, while still allowing entirely unannotated definitions.

For example, it would report an error for `def f(a: int, b)` but not `def f(a, b)`.

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.

1.22.9 None and Optional handling

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

implicit_optional

Type
boolean

Default
False

Causes mypy to treat parameters with a `None` default value as having an implicit optional type (`T | None`).

Note: This was True by default in mypy versions 0.980 and earlier.

strict_optional

Type
boolean

Default
True

Effectively disables checking of optional types and `None` values. With this option, mypy doesn't generally check the use of `None` values – it is treated as compatible with every type.

> **Warning**
>
> `strict_optional = false` is evil. Avoid using it and definitely do not use it without understanding what it does.

1.22.10 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.

`deprecated_calls_exclude`

Type
comma-separated list of strings

Selectively excludes functions and methods defined in specific packages, modules, and classes from the *deprecated* error code. This also applies to all submodules of packages (i.e. everything inside a given prefix). Note, this option does not support per-file configuration, the exclusions list is defined globally for all your code.

1.22.11 Suppressing errors

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

`ignore_errors`

Type
boolean

Default
False

Ignores all non-fatal errors.

1.22.12 Miscellaneous strictness flags

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

`allow_untyped_globals`

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

By default, mypy won't allow a variable to be redefined with an unrelated type. This flag enables the redefinition of unannotated variables with an arbitrary type. This also requires `local_partial_types`, which is enabled by default starting from mypy 2.0. Example:

```
def maybe_convert(n: int, b: bool) -> int | str:
    if b:
        x = str(n)  # Assign "str"
    else:
        x = n       # Assign "int"
    # Type of "x" is "int | str" here.
    return x
```

This also enables an unannotated variable to have different types in different code locations:

```
if check():
    for x in range(n):
        # Type of "x" is "int" here.
        ...
else:
    for x in ['a', 'b']:
        # Type of "x" is "str" here.
        ...
```

Function arguments are special, changing their type within function body is allowed even if they are annotated, but that annotation is used to infer types of r.h.s. of all subsequent assignments. Such middle-ground semantics provides good balance for majority of common use cases. For example:

```
def process(values: list[float]) -> None:
    if not values:
        values = [0, 0, 0]
    reveal_type(values)  # Revealed type is list[float]
```

Note: We are planning to turn this flag on by default in a future mypy release.

`allow_redefinition_new`

Type
boolean

Default
False

Deprecated alias for *allow_redefinition*.

allow_redefinition_old

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:

```
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

Type
boolean

Default
True

This prevents inferring a variable type from an empty container (such as a list or a dictionary) created at module top level or class body and updated in a function. This must be 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.

extra_checks

Type
boolean

Default
False

This flag enables additional checks that are technically correct but may be impractical. See *mypy --extra-checks* for more info.

`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:

```
# 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_equality`

Type
boolean

Default
False

Prohibit equality checks, identity checks, and container checks between non-overlapping types (except `None`).

`strict_equality_for_none`

Type
boolean

Default
False

Include `None` in strict equality checks (requires *strict_equality* to be activated).

`strict_bytes`

Type
boolean

Default
True

If disabled, mypy treats `bytearray` and `memoryview` as subtypes of `bytes`. This has been enabled by default since mypy 2.0.

`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.

1.22.13 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_code_links`

Type
boolean

Default
False

Shows documentation link to corresponding error code.

`hide_error_codes`

Type
boolean

Default
False

Hides 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.

1.22.14 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
True

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.

1.22.15 Parallel type-checking configuration

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

num_workers

Type
integer

Default
0

Use specific number of parallel worker processes for type-checking, see *parallel type-checking* for more details. This setting will be overridden by the `MYPY_NUM_WORKERS` environment variable.

1.22.16 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

This 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.

User home directory and environment variables will be expanded.

Note that this doesn't affect third-party library stubs. To test third-party stubs, for example try `MYPYPATH=stubs/six mypy ...`.

`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`*.

`native_parser`

Type
boolean

Default
False

This enables fast Rust-based parser that parses directly to mypy AST. It will become the default parser in one of the next mypy releases.

1.22.17 Report generation

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

> **Warning**
>
> Generating reports disables incremental mode and can significantly slow down your workflow. It is recommended to enable reporting only for specific runs (e.g. in CI).

`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]`.

1.22.18 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.

junit_format

Type
string

Default
`global`

If junit_xml is set, specifies format. global (default): single test with all errors; per_file: one test entry per file with failures.

scripts_are_modules

Type
boolean

Default
False

Makes script `x` become module `x` instead of `__main__`. This is useful when checking multiple scripts in a single run.

warn_unused_configs

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 *incremental = False*.)

verbosity

Type
integer

Default
0

Controls how much debug output will be generated. Higher numbers are more verbose.

1.22.19 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.mypy.overrides]]` sections:
  - For example, `[mypy-packagename]` would become:

```
[[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:

```
[[tool.mypy.overrides]]
module = [
    'packagename',
    'packagename2'
]
...
```

- 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.

1.22.20 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.

```
# mypy global options:

[tool.mypy]
python_version = "3.10"
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.mypy.overrides]]
module = "mycode.foo.*"
disallow_untyped_defs = true

[[tool.mypy.overrides]]
module = "mycode.bar"
warn_return_any = false

[[tool.mypy.overrides]]
module = [
    "somelibrary",
    "some_other_library"
]
ignore_missing_imports = true
```

1.23 Inline configuration

Mypy supports setting per-file configuration options inside files themselves using `# mypy:` comments. For example:

```
# mypy: disallow-any-generics
```

Inline configuration comments take precedence over all other configuration mechanisms.

1.23.1 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:

```
# 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:

```
# mypy: disallow-untyped-defs, always-false="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):

```
# mypy: allow-untyped-defs, no-strict-optional
```

1.24 Mypy daemon (mypy server)

Instead of running mypy as a command-line tool, you can also run it as a long-running daemon (server) process and use a command-line client to send type-checking requests to the server. This way mypy can perform type checking much faster, since program state cached from previous runs is kept in memory and doesn't have to be read from the file system on each run. The server also uses finer-grained dependency tracking to reduce the amount of work that needs to be done.

If you have a large codebase to check, running mypy using the mypy daemon can be *10 or more times faster* than the regular command-line `mypy` tool, especially if your workflow involves running mypy repeatedly after small edits – which is often a good idea, as this way you'll find errors sooner.

Note

The 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.

1.24.1 Basic usage

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

```
dmypy run -- 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.

> **Note**
>
> The mypy daemon requires `--local-partial-types`, which is enabled by default starting from mypy 2.0.

1.24.2 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.

1.24.3 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.

1.24.4 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:

```
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:

```
[{"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).

1.24.5 Statically inspect expressions

The daemon allows one to get the declared or inferred type of an expression (or other information about an expression, such as known attributes or definition location) using the 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:

```
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*.

--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.

--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").

--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

```
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".

--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.

--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"`.

--include-object-attrs

This will make the daemon include attributes of `object` (excluded by default) in case of an `atts` inspection.

--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:

```
from typing import Union

class A:
    x: int
    z: int
class B:
    y: int
    z: int
var: Union[A, B]
var
```

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).

1.25 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.

1.25.1 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 installed mypy. Alternatively, you can use the *--python-executable* flag to point to the Python executable for another environment, 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).

1.25.2 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:

```
setup.py
package_a/
    __init__.py
    lib.py
    py.typed
```

The `setup.py` file could look like this:

```
from setuptools import setup

setup(
    name="SuperPackageA",
    author="Me",
    version="0.1",
    package_data={"package_a": ["py.typed"]},
    packages=["package_a"]
)
```

Some packages have a mix of stub files and runtime files. These packages also require a `py.typed` file. An example can be seen below:

```
setup.py
package_b/
    __init__.py
    lib.py
    lib.pyi
    py.typed
```

The `setup.py` file might look like this:

```
from setuptools import setup
```

(continues on next page)

(continued from previous page)

```
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:

```
setup.py
package_c-stubs/
    __init__.pyi
    lib.pyi
```

The `setup.py` might look like this:

```
from setuptools 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`:

```
global-include *.pyi
global-include *.typed
```

1.26 Extending and integrating mypy

1.26.1 Integrating mypy into another Python application

It is possible to integrate mypy into another Python 3 application by importing `mypy.api` and calling the `run` function with a parameter of type `list[str]`, containing what normally would have been the command line arguments to mypy.

Function `run` returns a `tuple[str, str, int]`, namely `(<normal_report>, <error_report>, <exit_status>)`, in which `<normal_report>` is what mypy normally writes to `sys.stdout`, `<error_report>` is what mypy normally writes to `sys.stderr` and `exit_status` is the exit status mypy normally returns to the operating system.

A trivial example of using the api is the following

```python
import sys
from mypy import api

result = api.run(sys.argv[1:])

if result[0]:
    print('\nType checking report:\n')
    print(result[0])  # stdout

if result[1]:
    print('\nError report:\n')
    print(result[1])  # stderr

print('\nExit status:', result[2])
```

1.26.2 Extending mypy using plugins

Python is a highly dynamic language and has extensive metaprogramming capabilities. Many popular libraries use these to create APIs that may be more flexible and/or natural for humans, but are hard to express using static types. Extending the **PEP 484** type system to accommodate all existing dynamic patterns is impractical and often just impossible.

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

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

Note

The plugin system is experimental and prone to change. If you want to write a mypy plugin, we recommend you start by contacting the mypy core developers on gitter. In particular, there are no guarantees about backwards compatibility.

Backwards incompatible changes may be made without a deprecation period, but we will announce them in the plugin API changes announcement issue.

1.26.3 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:

```ini
[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:

```ini
[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.

1.26.4 High-level overview

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

```
from mypy.plugin import Plugin

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

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

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

1.26.5 Current list of plugin hooks

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

```
from lib import Vector

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

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

get_function_hook() is used to adjust the return type of a function call. This hook will be also called for instantiation of classes. This is a good choice if the return type is too complex to be expressed by regular python typing.

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 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:
    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:

```
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.

1.26.6 Useful tools

Mypy ships `mypy.plugins.proper_plugin` plugin which can be useful for plugin authors, since it finds missing `get_proper_type()` calls, which is a pretty common mistake.

It is recommended to enable it as a part of your plugin's CI.

1.27 Automatic stub generation (stubgen)

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

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

```
from other_module import dynamic

BORDER_WIDTH = 15

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

```
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.

1.27.1 Specifying what to stub

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

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

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

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

```
$ stubgen my_pkg_dir
```

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

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

Details of the options:

-m MODULE, **--module** MODULE

Generate a stub file for the given module. This flag may be repeated multiple times.

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

-p PACKAGE, **--package** PACKAGE

Generate stubs for the given package. This flag maybe repeated multiple times.

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

Note

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

Stubgen applies heuristics to avoid generating stubs for submodules that include tests or vendored third-party packages.

1.27.2 Specifying how to generate stubs

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

--no-import

Don't try to import modules. Instead 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.

--no-analysis

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. Does not apply to C extension modules. Incompatible with *--inspect-mode*.

--inspect-mode

Import and inspect modules instead of parsing source code. This is the default behavior for C modules and pyc-only packages. The flag is useful to force inspection for pure Python modules that make use of dynamically generated members that would otherwise be omitted when using the default behavior of code parsing. Implies *--no-analysis* as analysis requires source code.

--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.

1.27.3 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.

--include-docstrings

Include docstrings in stubs. This will add docstrings to Python function and classes stubs and to C extension function stubs.

--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.

1.28 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.

1.28.1 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.

> **Warning**
>
> stubtest will import and execute Python code from the packages it checks.

1.28.2 Example

Here's a quick example of what stubtest can do:

```
$ 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␣
↪argument does not
Stub: at line 3
def (x: builtins.int)
Runtime: in file ~/library.py:3
def (x=None)

error: library.x variable differs from runtime type Literal['hello, stubtest']
Stub: at line 1
builtins.int
Runtime:
'hello, stubtest'
```

1.28.3 Usage

Running stubtest can be as simple as `stubtest module_to_check`. Run *`stubtest --help`* for a quick summary of options.

Stubtest 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.

1.28.4 Allowlist

If you wish to ignore some of stubtest's complaints, stubtest supports a pretty handy *--allowlist* system.

Let's say that you have this python module called `ex`:

```
try:
    import optional_expensive_dep
except ImportError:
    optional_expensive_dep = None

first = 1
if optional_expensive_dep:
    second = 2
```

Let's say that you can't install `optional_expensive_dep` in CI for some reason, but you still want to include `second: int` in the stub file:

```
first: int
second: int
```

In this case stubtest will correctly complain:

```
error: ex.second is not present at runtime
Stub: in file /.../ex.pyi:2
builtins.int
Runtime:
MISSING

Found 1 error (checked 1 module)
```

To fix this, you can add an `allowlist` entry:

```
# Allowlist entries in `allowlist.txt` file:

# Does not exist if `optional_expensive_dep` is not installed:
ex.second
```

And now when running stubtest with `--allowlist=allowlist.txt`, no errors will be generated anymore.

Allowlists also support regular expressions, which can be useful to ignore many similar errors at once. They can also be useful for suppressing stubtest errors that occur sometimes, but not on every CI run. For example, if some CI workers have `optional_expensive_dep` installed, stubtest might complain with this message on those workers if you had the `ex.second` allowlist entry:

```
note: unused allowlist entry ex.second
Found 1 error (checked 1 module)
```

Changing `ex.second` to be `(ex\.second)?` will make this error optional, meaning that stubtest will pass whether or not a CI runner has``optional_expensive_dep`` installed.

1.28.5 CLI

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

--strict-type-check-only

Require `@type_check_only` on private types that are not present at runtime.

--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.

The presence of an entry in the allowlist means stubtest will not generate any errors for the corresponding definition.

--generate-allowlist

Print an allowlist (to stdout) to be used with *--allowlist*.

When introducing stubtest to an existing project, this is an easy way to silence all existing errors.

--ignore-unused-allowlist

Ignore unused allowlist entries

Without this option enabled, the default is for stubtest to complain if an allowlist entry is not necessary for stubtest to pass successfully.

Note if an allowlist entry is a regex that matches the empty string, stubtest will never consider it unused. For example, to get `--ignore-unused-allowlist` behaviour for a single allowlist entry like `foo.bar` you could add an allowlist entry `(foo\.bar)?`. This can be useful when an error only occurs on a specific platform.

--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 :-)

1.29 Common issues and solutions

This section has examples of cases when you need to update your code to use static typing, and ideas for working around issues if mypy doesn't work as expected. Statically typed code is often identical to normal Python code (except for type annotations), but sometimes you need to do things slightly differently.

1.29.1 No errors reported for obviously wrong code

There are several common reasons why obviously wrong code is not flagged as an error.

The function containing the error is not annotated.

Functions that do not have any annotations (neither for any argument nor for the return type) are not type-checked, and even the most blatant type errors (e.g. `2 + 'a'`) pass silently. The solution is to add annotations. Where that isn't possible, functions without annotations can be checked using *--check-untyped-defs*.

Example:

```
def foo(a):
    return '(' + a.split() + ')'  # No error!
```

This gives no error even though `a.split()` is "obviously" a list (the author probably meant `a.strip()`). The error is reported once you add annotations:

```
def foo(a: str) -> str:
    return '(' + a.split() + ')'
# error: Unsupported operand types for + ("str" and "list[str]")
```

If you don't know what types to add, you can use `Any`, but beware:

One of the values involved has type 'Any'.

Extending the above example, if we were to leave out the annotation for `a`, we'd get no error:

```
def foo(a) -> str:
    return '(' + a.split() + ')'  # No error!
```

The reason is that if the type of `a` is unknown, the type of `a.split()` is also unknown, so it is inferred as having type `Any`, and it is no error to add a string to an `Any`.

If you're having trouble debugging such situations, *reveal_type()* might come in handy.

Note that sometimes library stubs with imprecise type information can be a source of `Any` values.

`__init__` **method has no annotated arguments and no return type annotation.**

This is basically a combination of the two cases above, in that `__init__` without annotations can cause `Any` types leak into instance variables:

```
class Bad:
    def __init__(self):
        self.value = "asdf"
        1 + "asdf"  # No error!

bad = Bad()
bad.value + 1           # No error!
reveal_type(bad)        # Revealed type is "__main__.Bad"
reveal_type(bad.value)  # Revealed type is "Any"

class Good:
    def __init__(self) -> None:  # Explicitly return None
        self.value = value
```

Some imports may be silently ignored.

A common source of unexpected `Any` values is the *--ignore-missing-imports* flag.

When you use *--ignore-missing-imports*, any imported module that cannot be found is silently replaced with Any.

To help debug this, simply leave out *--ignore-missing-imports*. As mentioned in *Missing imports*, setting ignore_missing_imports=True on a per-module basis will make bad surprises less likely and is highly encouraged.

Use of the *--follow-imports=skip* flags can also cause problems. Use of these flags is strongly discouraged and only required in relatively niche situations. See *Following imports* for more information.

mypy considers some of your code unreachable.

See *Unreachable code* for more information.

A function annotated as returning a non-optional type returns 'None' and mypy doesn't complain.

```
def foo() -> str:
    return None  # No error!
```

You may have disabled strict optional checking (see *–no-strict-optional* for more).

1.29.2 Spurious errors and locally silencing the checker

You can use a # type: ignore comment to silence the type checker on a particular line. For example, let's say our code is using the C extension module frobnicate, and there's no stub available. Mypy will complain about this, as it has no information about the module:

```
import frobnicate  # Error: No module "frobnicate"
frobnicate.start()
```

You can add a # type: ignore comment to tell mypy to ignore this error:

```
import frobnicate  # type: ignore
frobnicate.start()  # Okay!
```

The second line is now fine, since the ignore comment causes the name frobnicate to get an implicit Any type.

The type ignore comment must be at the start of the comments on a line. This type ignore will not take effect:

```
import frobnicate  #example other comment # type: ignore
frobnicate.start()
```

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__`:

```
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.

1.29.3 Ignoring a whole file

- To only ignore errors, use a top-level `# mypy: ignore-errors` comment instead.
- To only ignore errors with a specific error code, use a top-level `# mypy: disable-error-code="..."` comment. Example: `# mypy: disable-error-code="truthy-bool, ignore-without-code"`
- To replace the contents of a module with `Any`, use a per-module `follow_imports = skip`. See *Following imports* for details.

Note that a `# type: ignore` comment at the top of a module (before any statements, including imports or docstrings) has the effect of ignoring the entire contents of the module. This behaviour can be surprising and result in "Module … has no attribute … [attr-defined]" errors.

1.29.4 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*.

1.29.5 Mypy runs are slow

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

Furthermore: as of mypy 1.13, mypy allows use of the orjson library for handling the cache instead of the stdlib json, for improved performance. You can ensure the presence of orjson using the `faster-cache` extra:

> python3 -m pip install -U mypy[faster-cache]

Mypy may depend on orjson by default in the future.

1.29.6 Types of empty collections

You often need to specify the type when you assign an empty list or dict to a new variable, as mentioned earlier:

```
a: list[int] = []
```

Without the annotation mypy can't always figure out the precise type of a.

You can use a simple empty list literal in a dynamically typed function (as the type of a would be implicitly Any and need not be inferred), if type of the variable has been declared or inferred before, or if you perform a simple modification operation in the same scope (such as append for a list):

```
a = []  # Okay because followed by append, inferred type list[int]
for i in range(n):
    a.append(i * i)
```

However, in more complex cases an explicit type annotation can be required (mypy will tell you this). Often the annotation can make your code easier to understand, so it doesn't only help mypy but everybody who is reading the code!

1.29.7 Redefinitions with incompatible types

Each name within a function only has a single 'declared' type. You can reuse for loop indices etc., but if you want to use a variable with multiple types within a single function, you may need to instead use multiple variables (or maybe declare the variable with an Any type).

```
def f() -> None:
    n = 1
    ...
    n = 'x'  # error: Incompatible types in assignment (expression has type "str",␣
↪variable has type "int")
```

Note

Using the *--allow-redefinition* flag can suppress this error in several cases.

Note that you can redefine a variable with a more *precise* or a more concrete type. For example, you can redefine a sequence (which does not support sort()) as a list and sort it in-place:

```
def f(x: Sequence[int]) -> None:
    # Type of x is Sequence[int] here; we don't know the concrete type.
    x = list(x)
    # Type of x is list[int] here.
    x.sort()  # Okay!
```

See *Type narrowing* for more information.

1.29.8 Invariance vs covariance

Most mutable generic collections are invariant. When using the legacy TypeVar syntax, mypy considers all user-defined generic classes invariant by default (see *Variance of generic types* for motivation). When using the **PEP 695** syntax (class MyClass[T]: ...), variance is inferred from usage rather than defaulting to invariant.

The fact that mutable sequences are usually invariant can lead to some unexpected errors when combined with type inference. For example:

```
class A: ...
class B(A): ...

lst = [A(), A()]  # Inferred type is list[A]
```

(continues on next page)

(continued from previous page)

```
new_lst = [B(), B()]  # inferred type is list[B]
lst = new_lst  # mypy will complain about this, because List is invariant
```

Possible strategies in such situations are:

- Use an explicit type annotation:

```
new_lst: list[A] = [B(), B()]
lst = new_lst  # OK
```

- Make a copy of the right hand side:

```
lst = list(new_lst) # Also OK
```

- Use immutable collections as annotations whenever possible:

```
def f_bad(x: list[A]) -> A:
    return x[0]
f_bad(new_lst) # Fails

def f_good(x: Sequence[A]) -> A:
    return x[0]
f_good(new_lst) # OK
```

1.29.9 Declaring a supertype as variable type

Sometimes the inferred type is a subtype (subclass) of the desired type. The type inference uses the first assignment to infer the type of a name:

```
class Shape: ...
class Circle(Shape): ...
class Triangle(Shape): ...

shape = Circle()    # mypy infers the type of shape to be Circle
shape = Triangle()  # error: Incompatible types in assignment (expression has type
↪"Triangle", variable has type "Circle")
```

You can just give an explicit type for the variable in cases such the above example:

```
shape: Shape = Circle()  # The variable s can be any Shape, not just Circle
shape = Triangle()       # OK
```

1.29.10 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 collections.abc import Sequence
from typing import 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:
```

(continues on next page)

(continued from previous page)

```
        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:

```
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.

1.29.11 Python version and system platform checks

Mypy supports the ability to perform Python version checks and platform checks (e.g. Windows vs Posix), ignoring code paths that won't be run on the targeted Python version or platform. This allows you to more effectively typecheck code that supports multiple versions of Python or multiple operating systems.

More specifically, mypy will understand the use of `sys.version_info` and `sys.platform` checks within `if/elif/else` statements. For example:

```
import sys

# Distinguishing between different versions of Python:
if sys.version_info >= (3, 13):
    # Python 3.13+ specific definitions and imports
else:
    # Other definitions and imports

# Distinguishing between different operating systems:
if sys.platform.startswith("linux"):
```

(continues on next page)

(continued from previous page)

```
    # Linux-specific code
elif sys.platform == "darwin":
    # Mac-specific code
elif sys.platform == "win32":
    # Windows-specific code
else:
    # Other systems
```

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

```
import sys

assert sys.platform != 'win32'

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

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

Note

Mypy currently does not support more complex checks, and does not assign any special meaning when assigning a `sys.version_info` or `sys.platform` check to a variable. This may change in future versions of mypy.

By default, mypy will use your current version of Python and your current operating system as default values for `sys.version_info` and `sys.platform`.

To target a different Python version, use the *--python-version X.Y* flag. For example, to verify your code typechecks if were run using Python 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 typechecks if it were run in Windows, pass in *--platform win32*. See the documentation for `sys.platform` for examples of valid platform parameters.

1.29.12 Displaying the type of an expression

You can use `reveal_type(expr)` to ask mypy to display the inferred static type of an expression. This can be useful when you don't quite understand how mypy handles a particular piece of code. Example:

```
reveal_type((1, 'hello'))  # Revealed type is "tuple[builtins.int, builtins.str]"
```

You can also use `reveal_locals()` at any line in a file to see the types of all local variables at once. Example:

```
a = 1
b = 'one'
reveal_locals()
# Revealed local types are:
#     a: builtins.int
#     b: builtins.str
```

Note

reveal_type and reveal_locals are handled specially by mypy during type checking, and don't have to be defined or imported.

However, if you want to run your code, you'll have to remove any reveal_type and reveal_locals calls from your program or else Python will give you an error at runtime.

Alternatively, you can import reveal_type from typing_extensions or typing (on Python 3.11 and newer)

1.29.13 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
```

1.29.14 Covariant subtyping of mutable protocol members is rejected

Mypy rejects this because this is potentially unsafe. Consider this example:

```
from typing 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 import Protocol

class P(Protocol):
    @property
    def x(self) -> float:
       pass

def fun(arg: P) -> None:
    ...
```

(continues on next page)

(continued from previous page)

```
class C:
    x = 42
fun(C())  # OK
```

1.29.15 Dealing with conflicting names

Suppose you have a class with a method whose name is the same as an imported (or built-in) type, and you want to use the type in another method signature. E.g.:

```
class Message:
    def bytes(self):
        ...
    def register(self, path: bytes):  # error: Invalid type "mod.Message.bytes"
        ...
```

The third line elicits an error because mypy sees the argument type `bytes` as a reference to the method by that name. Other than renaming the method, a workaround is to use an alias:

```
bytes_ = bytes
class Message:
    def bytes(self):
        ...
    def register(self, path: bytes_):
        ...
```

1.29.16 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
python3 -m pip install --upgrade .
```

To install a development version of mypy that is mypyc-compiled, see the instructions at the mypyc wheels repo.

1.29.17 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:

   ```
   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:

   ```
   class A: ...
   Alias = A
   ```

   You can also use `TypeAlias` (**PEP 613**) to define an *explicit type alias*:

```
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*):

```
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
```

1.29.18 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:

```
from collections.abc import Sequence, 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):
```

(continues on next page)

(continued from previous page)

```
    # 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]
        ...
```

1.29.19 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:

```
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 mistyped code below it. For a more subtle example, consider this code:

```
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:

```
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.

1.29.20 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:

```
def foo(x: int | None) -> 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

    x = None  # because x could later be assigned None
    return inner

inner = foo(5)
inner()  # this will raise an error when called
```

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.

1.29.21 Incorrect use of `Self`

`Self` is not the type of the current class; it's a type variable with upper bound of the current class. That is, it represents the type of the current class or of potential subclasses.

```
from typing import Self

class Foo:
    @classmethod
    def constructor(cls) -> Self:
        # Instead, either call cls() or change the annotation to -> Foo
        return Foo()  # error: Incompatible return value type (got "Foo", expected "Self
↪")

class Bar(Foo):
    ...

reveal_type(Foo.constructor())  # note: Revealed type is "Foo"
# In the context of the subclass Bar, the Self return type promises
# that the return value will be Bar
reveal_type(Bar.constructor())  # note: Revealed type is "Bar"
```

1.30 Supported Python features

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

- Unsupported Python features

1.30.1 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.

1.31 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.

1.31.1 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.

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]
```

1.31.2 Enabling/disabling specific error codes globally

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* and *--disable-error-code* to enable or disable specific error codes that don't have a dedicated command-line flag or config file setting.

1.31.3 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
```

(continues on next page)

(continued from previous page)

```
[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: disable-error-code="..."` and `# mypy: enable-error-code="..."` comments can further *adjust* them for a specific file. For example:

```
# mypy: enable-error-code="truthy-bool, ignore-without-code"
```

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.

1.31.4 Subcodes of error codes

In some cases, mostly for backwards compatibility reasons, an error code may be covered also by another, wider error code. For example, an error with code `[method-assign]` can be ignored by `# type: ignore[assignment]`. Similar logic works for disabling error codes globally. If a given error code is a subcode of another one, it will be mentioned in the documentation for the narrower code. This hierarchy is not nested: there cannot be subcodes of other subcodes.

1.31.5 Requiring error codes

It's possible to require error codes be specified in `type: ignore` comments. See *ignore-without-code* for more information.

1.32 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.

1.32.1 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:

```
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):

```
# 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.

1.32.2 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:

```
class Cat:
    def sleep(self) -> None: ...
    def miaow(self) -> None: ...

class Dog:
    def sleep(self) -> None: ...
    def follow_me(self) -> None: ...

def func(animal: Cat | Dog) -> None:
    # OK: 'sleep' is defined for both Cat and Dog
    animal.sleep()
    # Error: Item "Cat" of "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.

1.32.3 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()`:

```
x = sort([3, 2, 4])  # Error: Name "sort" is not defined  [name-defined]
```

1.32.4 Check that a variable is not used before it's defined [used-before-def]

Mypy will generate an error if a name is used before it's defined. While the name-defined check will catch issues with names that are undefined, it will not flag if a variable is used and then defined later in the scope. used-before-def check will catch such cases.

Example:

```
print(x)  # Error: Name "x" is used before definition [used-before-def]
x = 123
```

1.32.5 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:

```
def greet(name: str) -> None:
     print('hello', name)

greet('jack')  # OK
greet('jill', 'jack')  # Error: Too many arguments for "greet"  [call-arg]
```

1.32.6 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:

```
def first(x: list[int]) -> 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))
```

1.32.7 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:

```
from typing import overload

@overload
def inc_maybe(x: None) -> None: ...

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

def inc_maybe(x: int | None) -> int | None:
```

(continues on next page)

(continued from previous page)

```
    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)
```

1.32.8 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:

```
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:

```
from collections.abc import Callable

# OK
def log_all(objs: list[object], f: Callable[[object], None]) -> None:
    for x in objs:
        f(x)
```

1.32.9 Check that NoneType is not used as a type (annotation) [nonetype-type]

The preferred way to annotate the type of *None* is *None*. *NoneType* is equivalent, but mypy won't allow it by default.

```
from types import NoneType
def f(x: None) -> None:
    reveal_type(x) # note: Revealed type is "None"

# error: NoneType should not be used as a type, please use None instead  [nonetype-type]
def g(x: NoneType) -> None:
    reveal_type(x) # note: Revealed type is "None"

# error: NoneType should not be used as a type, please use None instead  [nonetype-type]
x1: NoneType = None
x2: None = None # OK
```

1.32.10 Check the validity of a class's metaclass [metaclass]

Mypy checks whether the metaclass of a class is valid. The metaclass must be a subclass of `type`. Further, the class hierarchy must yield a consistent metaclass. For more details, see the Python documentation

Note that mypy's metaclass checking is limited and may produce false-positives. See also *Gotchas and limitations of metaclass support*.

Example with an error:

```
class GoodMeta(type):
    pass

class BadMeta:
    pass

class A1(metaclass=GoodMeta):  # OK
    pass

class A2(metaclass=BadMeta):  # Error:  Metaclasses not inheriting from "type" are not␣
↪supported  [metaclass]
    pass
```

1.32.11 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]
```

1.32.12 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:

```
class Base:
    def method(self,
               arg: int) -> int | None:
        ...

class Derived(Base):
    def method(self,
               arg: int | str) -> int:  # OK
        ...

class DerivedBad(Base):
    # Error: Argument 1 of "method" is incompatible with "Base"  [override]
    def method(self,
               arg: bool) -> int:
        ...
```

1.32.13 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:

```
# 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')
```

1.32.14 Check that functions don't have empty bodies outside stubs [empty-body]

This error code is similar to the `[return]` code but is emitted specifically for functions and methods with empty bodies (if they are annotated with non-trivial return type). Such a distinction exists because in some contexts an empty body can be valid, for example for an abstract method or in a stub file. Also old versions of mypy used to unconditionally allow functions with empty bodies, so having a dedicated error code simplifies cross-version compatibility.

Note that empty bodies are allowed for methods in *protocols*, and such methods are considered implicitly abstract:

```
from abc import abstractmethod
from typing import Protocol

class RegularABC:
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def foo(self) -> int:
        pass  # OK
    def bar(self) -> int:
        pass  # Error: Missing return statement  [empty-body]

class Proto(Protocol):
    def bar(self) -> int:
        pass  # OK
```

1.32.15 Check that return value is compatible [return-value]

Mypy checks that the returned value is compatible with the type signature of the function.

Example:

```
def func(x: int) -> str:
    # Error: Incompatible return value type (got "int", expected "str")  [return-value]
    return x + 1
```

1.32.16 Check types in assignment statement [assignment]

Mypy checks that the assigned expression is compatible with the assignment target (or targets).

Example:

```
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
```

1.32.17 Check that assignment target is not a method [method-assign]

In general, assigning to a method on class object or instance (a.k.a. monkey-patching) is ambiguous in terms of types, since Python's static type system cannot express the difference between bound and unbound callable types. Consider this example:

```
class A:
    def f(self) -> None: pass
    def g(self) -> None: pass

def h(self: A) -> None: pass

A.f = h  # Type of h is Callable[[A], None]
A().f()  # This works
```

(continues on next page)

(continued from previous page)

```
A.f = A().g  # Type of A().g is Callable[[], None]
A().f()  # ...but this also works at runtime
```

To prevent the ambiguity, mypy will flag both assignments by default. If this error code is disabled, mypy will treat the assigned value in all method assignments as unbound, so only the second assignment will still generate an error.

Note

This error code is a subcode of the more general `[assignment]` code.

1.32.18 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 3.12 syntax):

```
def add[T1: (int, float)](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')
```

1.32.19 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:

```
# Error: Unsupported operand types for + ("int" and "str")  [operator]
1 + 'x'
```

1.32.20 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:

```
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
```

1.32.21 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]
```

1.32.22 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'}
```

1.32.23 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 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}
```

1.32.24 Check TypedDict Keys [typeddict-unknown-key]

When constructing a TypedDict object, mypy checks whether the definition contains unknown keys, to catch invalid keys and misspellings. On the other hand, mypy will not generate an error when a previously constructed TypedDict value with extra keys is passed to a function as an argument, since TypedDict values support structural subtyping ("static duck typing") and the keys are assumed to have been validated at the point of construction. Example:

```
from typing import TypedDict

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

class Point3D(Point):
```

(continues on next page)

(continued from previous page)

```
    z: int

def add_x_coordinates(a: Point, b: Point) -> int:
    return a["x"] + b["x"]

a: Point = {"x": 1, "y": 4}
b: Point3D = {"x": 2, "y": 5, "z": 6}

add_x_coordinates(a, b)  # OK

# Error: Extra key "z" for TypedDict "Point"  [typeddict-unknown-key]
add_x_coordinates(a, {"x": 1, "y": 4, "z": 5})
```

Setting a TypedDict item using an unknown key will also generate this error, since it could be a misspelling:

```
a: Point = {"x": 1, "y": 2}
# Error: Extra key "z" for TypedDict "Point"  [typeddict-unknown-key]
a["z"] = 3
```

Reading an unknown key will generate the more general (and serious) `typeddict-item` error, which is likely to result in an exception at runtime:

```
a: Point = {"x": 1, "y": 2}
# Error: TypedDict "Point" has no key "z"  [typeddict-item]
_ = a["z"]
```

Note

This error code is a subcode of the wider `[typeddict-item]` code.

1.32.25 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]
        self.x = self.y

    def set_y(self) -> None:
        self.y = self.x
```

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 `y` attribute to work around the issue:

```
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
```

1.32.26 Check for an issue with imports [import]

Mypy generates an error if it can't resolve an *import* statement. This is a parent error code of *import-not-found* and *import-untyped*

See *Missing imports* for how to work around these errors.

1.32.27 Check that import target can be found [import-not-found]

Mypy generates an error if it can't find the source code or a stub file for an imported module.

Example:

```
# Error: Cannot find implementation or library stub for module named "m0dule_with_typo"␣
↪[import-not-found]
import m0dule_with_typo
```

See *Missing imports* for how to work around these errors.

1.32.28 Check that import target can be found [import-untyped]

Mypy generates an error if it can find the source code for an imported module, but that module does not provide type annotations (via *PEP 561*).

Example:

```
# Error: Library stubs not installed for "bs4"  [import-untyped]
import bs4
# Error: Skipping analyzing "no_py_typed": module is installed, but missing library␣
↪stubs or py.typed marker  [import-untyped]
import no_py_typed
```

In some cases, these errors can be fixed by installing an appropriate stub package. See *Missing imports* for more details.

1.32.29 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:

```
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: ...
```

(continues on next page)

(continued from previous page)

```
# Error: Argument 1 to "A" has incompatible type "str"; expected "int"
#        (the first definition wins!)
A('x')
```

1.32.30 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`:

```
def f() -> None:
    ...

# OK: we don't do anything with the return value
f()

# Error: "f" does not return a value (it only ever returns None)  [func-returns-value]
if f():
     print("not false")
```

1.32.31 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:

```
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" ␣
↪[abstract]
t = Thing()
```

1.32.32 Safe handling of abstract type object types [type-abstract]

Mypy always allows instantiating (calling) type objects typed as `type[t]`, even if it is not known that `t` is non-abstract, since it is a common pattern to create functions that act as object factories (custom constructors). Therefore, to prevent issues described in the above section, when an abstract type object is passed where `type[t]` is expected, mypy will give an error. Example (Python 3.12 syntax):

```
from abc import ABCMeta, abstractmethod

class Config(metaclass=ABCMeta):
    @abstractmethod
    def get_value(self, attr: str) -> str: ...

def make_many[T](typ: type[T], n: int) -> list[T]:
    return [typ() for _ in range(n)]  # This will raise if typ is abstract

# Error: Only concrete class can be given where "type[Config]" is expected [type-
↪abstract]
make_many(Config, 5)
```

1.32.33 Check that call to an abstract method via super is valid [safe-super]

Abstract methods often don't have any default implementation, i.e. their bodies are just empty. Calling such methods in subclasses via `super()` will cause runtime errors, so mypy prevents you from doing so:

```
from abc import abstractmethod
class Base:
    @abstractmethod
    def foo(self) -> int: ...
class Sub(Base):
    def foo(self) -> int:
        return super().foo() + 1  # error: Call to abstract method "foo" of "Base" with
                                  # trivial body via super() is unsafe  [safe-super]
Sub().foo()  # This will crash at runtime.
```

Mypy considers the following as trivial bodies: a `pass` statement, a literal ellipsis `...`, a docstring, and a `raise NotImplementedError` statement.

1.32.34 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:

```
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.

1.32.35 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.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:

```
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␣
↪False
example.py:3: note: Use "typing_extensions.Literal[False]" as the return type or change␣
↪it to
    "None"
example.py:3: note: If return type of "__exit__" implies that it may return True, the␣
↪context
    manager may swallow exceptions
```

You can use `Literal[False]` to fix the error:

```
from typing import Literal

class MyContext:
    ...
    def __exit__(self, exc, value, tb) -> Literal[False]:  # OK
        print('exit')
        return False
```

You can also use `None`:

```
class MyContext:
    ...
    def __exit__(self, exc, value, tb) -> None:  # Also OK
        print('exit')
```

1.32.36 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:

```
from typing import NamedTuple

# Error: First argument to namedtuple() should be "Point2D", not "Point"
Point2D = NamedTuple("Point", [("x", int), ("y", int)])
```

1.32.37 Check that literal is used where expected [literal-required]

There are some places where only a (string) literal value is expected for the purposes of static type checking, for example a `TypedDict` key, or a `__match_args__` item. Providing a `str`-valued variable in such contexts will result in an error. Note that in many cases you can also use `Final` or `Literal` variables. Example:

```
from typing import Final, Literal, TypedDict

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

def test(p: Point) -> None:
    X: Final = "x"
    p[X]  # OK

    Y: Literal["y"] = "y"
    p[Y]  # OK

    key = "x"  # Inferred type of key is `str`
    # Error: TypedDict key must be a string literal;
    #   expected one of ("x", "y")  [literal-required]
    p[key]
```

1.32.38 Check that overloaded functions have an implementation [no-overload-impl]

Overloaded functions outside of stub files must be followed by a non overloaded implementation.

```
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
```

1.32.39 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.

```
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:

```
_ = f()  # No error
```

1.32.40 Warn about top level await expressions [top-level-await]

This error code is separate from the general `[syntax]` errors, because in some environments (e.g. IPython) a top level `await` is allowed. In such environments a user may want to use `--disable-error-code=top-level-await`, which allows one to still have errors for other improper uses of `await`, for example:

```
async def f() -> None:
    ...

top = await f()  # Error: "await" outside function  [top-level-await]
```

1.32.41 Warn about await expressions used outside of coroutines [await-not-async]

`await` must be used inside a coroutine.

```
async def f() -> None:
    ...

def g() -> None:
    await f()  # Error: "await" outside coroutine ("async def")  [await-not-async]
```

1.32.42 Check types in assert_type [assert-type]

The inferred type for an expression passed to `assert_type` must match the provided type.

```
from typing_extensions import assert_type

assert_type([1], list[int])  # OK

assert_type([1], list[str])  # Error
```

1.32.43 Check that function isn't used in boolean context [truthy-function]

Functions will always evaluate to true in boolean contexts.

```
def f():
    ...

if f:  # Error: Function "Callable[[], Any]" could always be true in boolean context ␣
↪[truthy-function]
    pass
```

1.32.44 Check that string formatting/interpolation is type-safe [str-format]

Mypy will check that f-strings, `str.format()` calls, and `%` interpolations are valid (when corresponding template is a literal string). This includes checking number and types of replacements, for example:

```
# Error: Cannot find replacement for positional format specifier 1 [str-format]
"{} and {}".format("spam")
"{} and {}".format("spam", "eggs")  # OK
# Error: Not all arguments converted during string formatting [str-format]
"{} and {}".format("spam", "eggs", "cheese")

# Error: Incompatible types in string interpolation
# (expression has type "float", placeholder has type "int") [str-format]
"{:d}".format(3.14)
```

1.32.45 Check for implicit bytes coercions [str-bytes-safe]

Warn about cases where a bytes object may be converted to a string in an unexpected manner.

```
b = b"abc"

# Error: If x = b'abc' then f"{x}" or "{}".format(x) produces "b'abc'", not "abc".
# If this is desired behavior, use f"{x!r}" or "{!r}".format(x).
# Otherwise, decode the bytes [str-bytes-safe]
print(f"The alphabet starts with {b}")

# Okay
print(f"The alphabet starts with {b!r}")  # The alphabet starts with b'abc'
print(f"The alphabet starts with {b.decode('utf-8')}")  # The alphabet starts with abc
```

1.32.46 Check that `str` is not unpacked [str-unpack]

It can sometimes be surprising that `str` is iterable, especially when unpacking in an assignment.

Example:

```
def print_dict(d: dict[str, str]) -> int:
    # We meant to do d.items(), but instead we're unpacking the str keys of d

    # Error: Unpacking a string is disallowed
    for k, v in d:
        print(k, v)
```

1.32.47 Check that overloaded functions don't overlap [overload-overlap]

Warn if multiple `@overload` variants overlap in potentially unsafe ways. This guards against the following situation:

```
from typing import overload

class A: ...
class B(A): ...

@overload
def foo(x: B) -> int: ...  # Error: Overloaded function signatures 1 and 2 overlap with␣
↪incompatible return types  [overload-overlap]
@overload
def foo(x: A) -> str: ...
def foo(x): ...
```

(continues on next page)

(continued from previous page)

```
def takes_a(a: A) -> str:
    return foo(a)

a: A = B()
value = takes_a(a)
# mypy will think that value is a str, but it could actually be an int
reveal_type(value) # Revealed type is "builtins.str"
```

Note that in cases where you ignore this error, mypy will usually still infer the types you expect.

See *overloading* for more explanation.

1.32.48 Check for overload signatures that cannot match [overload-cannot-match]

Warn if an `@overload` variant can never be matched, because an earlier overload has a wider signature. For example, this can happen if the two overloads accept the same parameters and each parameter on the first overload has the same type or a wider type than the corresponding parameter on the second overload.

Example:

```
from typing import overload, Union

@overload
def process(response1: object, response2: object) -> object:
    ...
@overload
def process(response1: int, response2: int) -> int: # E: Overloaded function signature 2␣
↪will never be matched: signature 1's parameter type(s) are the same or broader ␣
↪[overload-cannot-match]
    ...

def process(response1: object, response2: object) -> object:
    return response1 + response2
```

1.32.49 Notify about an annotation in an unchecked function [annotation-unchecked]

Sometimes a user may accidentally omit an annotation for a function, and mypy will not check the body of this function (unless one uses *--check-untyped-defs* or *--disallow-untyped-defs*). To avoid such situations go unnoticed, mypy will show a note, if there are any type annotations in an unchecked function:

```
def test_assignment():  # "-> None" return annotation is missing
    # Note: By default the bodies of untyped functions are not checked,
    # consider using --check-untyped-defs [annotation-unchecked]
    x: int = "no way"
```

Note that mypy will still exit with return code `0`, since such behaviour is specified by **PEP 484**.

1.32.50 Decorator preceding property not supported [prop-decorator]

Mypy does not yet support analysis of decorators that precede the property decorator. If the decorator does not preserve the declared type of the property, mypy will not infer the correct type for the declaration. If the decorator cannot be moved after the `@property` decorator, then you must use a type ignore comment:

```
class MyClass:
    @special  # type: ignore[prop-decorator]
    @property
    def magic(self) -> str:
        return "xyzzy"
```

Note

For backward compatibility, this error code is a subcode of the generic `[misc]` code.

1.32.51 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.

1.32.52 ReadOnly key of a TypedDict is mutated [typeddict-readonly-mutated]

Consider this example:

```
from datetime import datetime
from typing import TypedDict
from typing_extensions import ReadOnly

class User(TypedDict):
    username: ReadOnly[str]
    last_active: datetime

user: User = {'username': 'foobar', 'last_active': datetime.now()}
user['last_active'] = datetime.now()  # ok
user['username'] = 'other'  # error: ReadOnly TypedDict key "key" TypedDict is mutated [typeddict-readonly-mutated]
```

PEP 705 specifies how `ReadOnly` special form works for `TypedDict` objects.

1.32.53 Check that `TypeIs` narrows types [narrowed-type-not-subtype]

PEP 742 requires that when `TypeIs` is used, the narrowed type must be a subtype of the original type:

```
from typing_extensions import TypeIs

def f(x: int) -> TypeIs[str]:  # Error, str is not a subtype of int
    ...

def g(x: object) -> TypeIs[str]:  # OK
    ...
```

1.32.54 String appears in a context which expects a TypeForm [maybe-unrecognized-str-typeform]

TypeForm literals may contain string annotations:

```
typx1: TypeForm = str | None
typx2: TypeForm = 'str | None'  # OK
typx3: TypeForm = 'str' | None  # OK
```

However TypeForm literals containing a string annotation can only be recognized by mypy in the following locations:

```
typx_var: TypeForm = 'str | None'  # assignment r-value

def func(typx_param: TypeForm) -> TypeForm:
    return 'str | None'  # returned expression

func('str | None')  # callable's argument
```

If you try to use a string annotation in some other location which expects a TypeForm, the string value will always be treated as a `str` even if a `TypeForm` would be more appropriate and this error code will be generated:

```
# Error: TypeForm containing a string annotation cannot be recognized here. Surround␣
↪with TypeForm(...) to recognize.  [maybe-unrecognized-str-typeform]
# Error: List item 0 has incompatible type "str"; expected "TypeForm[Any]"  [list-item]
list_of_typx: list[TypeForm] = ['str | None', float]
```

Fix the error by surrounding the entire type with `TypeForm(...)`:

```
list_of_typx: list[TypeForm] = [TypeForm('str | None'), float]  # OK
```

Similarly, if you try to use a string literal in a location which expects a TypeForm, this error code will be generated:

```
dict_of_typx = {'str_or_none': TypeForm(str | None)}
# Error: TypeForm containing a string annotation cannot be recognized here. Surround␣
↪with TypeForm(...) to recognize.  [maybe-unrecognized-str-typeform]
list_of_typx: list[TypeForm] = [dict_of_typx['str_or_none']]
```

Fix the error by adding `# type: ignore[maybe-unrecognized-str-typeform]` to the line with the string literal:

```
dict_of_typx = {'str_or_none': TypeForm(str | None)}
list_of_typx: list[TypeForm] = [dict_of_typx['str_or_none']]  # type: ignore[maybe-
↪unrecognized-str-typeform]
```

1.32.55 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.

1.33 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 and their configuration. *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*.

1.33.1 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:

```
# mypy: disallow-any-generics

# Error: Missing type arguments for generic type "list"  [type-arg]
def remove_dups(items: list) -> list:
    ...
```

1.33.2 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:

```
# 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
```

1.33.3 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:

```
# mypy: warn-redundant-casts

from typing import cast

Count = int

def example(x: Count) -> int:
    # Error: Redundant cast to "int"  [redundant-cast]
    return cast(int, x)
```

1.33.4 Check that methods do not have redundant Self annotations [redundant-self]

If a method uses the `Self` type in the return type or the type of a non-self argument, there is no need to annotate the `self` argument explicitly. Such annotations are allowed by **PEP 673** but are redundant. If you enable this error code, mypy will generate an error if there is a redundant `Self` type.

Example:

```
# mypy: enable-error-code="redundant-self"

from typing import Self

class C:
    # Error: Redundant "Self" annotation for the first method argument
    def copy(self: Self) -> Self:
        return type(self)()
```

1.33.5 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:

```
# 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:

```
# mypy: strict-equality

def is_magic(x: bytes) -> bool:
    return x == b'magic'  # OK
```

--strict-equality does not include comparisons with None:

```
# mypy: strict-equality

def is_none(x: str) -> bool:
    return x is None  # OK
```

If you want such checks, you must also activate *--strict-equality-for-none* (we might merge these two options later).

```
# mypy: strict-equality strict-equality-for-none

def is_none(x: str) -> bool:
    # Error: Non-overlapping identity check
    #        (left operand type: "str", right operand type: "None")
    return x is None
```

1.33.6 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:

```
# mypy: disallow-untyped-calls

def do_it() -> None:
    # Error: Call to untyped function "bad" in typed context  [no-untyped-call]
    bad()

def bad():
    ...
```

1.33.7 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:

```
# mypy: warn-return-any

def fields(s):
     return s.split(',')

def first_field(x: str) -> str:
    # Error: Returning Any from function declared to return "str"  [no-any-return]
    return fields(x)[0]
```

1.33.8 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:

```
# 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:
    ...
```

1.33.9 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.

```
# 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')
```

1.33.10 Check that imported or used feature is deprecated [deprecated]

If you use *--enable-error-code deprecated*, mypy generates an error if your code imports a deprecated feature explicitly with a `from mod import depr` statement or uses a deprecated feature imported otherwise or defined locally. Features are considered deprecated when decorated with `warnings.deprecated`, as specified in PEP 702. Use the *--report-deprecated-as-note* option to turn all such errors into notes. Use *--deprecated-calls-exclude* to hide warnings for specific functions, classes and packages.

> **Note**
>
> The `warnings` module provides the `@deprecated` decorator since Python 3.13. To use it with older Python versions, import it from `typing_extensions` instead.

Examples:

```
# mypy: report-deprecated-as-error

# Error: abc.abstractproperty is deprecated: Deprecated, use 'property' with
↪'abstractmethod' instead
from abc import abstractproperty

from typing_extensions import deprecated

@deprecated("use new_function")
```

(continues on next page)

(continued from previous page)

```
def old_function() -> None:
    print("I am old")

# Error: __main__.old_function is deprecated: use new_function
old_function()
old_function()  # type: ignore[deprecated]
```

1.33.11 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.

```
# 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)]
```

1.33.12 Warn about variables that are defined only in some execution paths [possibly-undefined]

If you use *--enable-error-code possibly-undefined*, mypy generates an error if it cannot verify that a variable will be defined in all execution paths. This includes situations when a variable definition appears in a loop, in a conditional branch, in an except handler, etc. For example:

```
# mypy: enable-error-code="possibly-undefined"

from collections.abc import Iterable

def test(values: Iterable[int], flag: bool) -> None:
    if flag:
        a = 1
    z = a + 1  # Error: Name "a" may be undefined [possibly-undefined]

    for v in values:
        b = v
    z = b + 1  # Error: Name "b" may be undefined [possibly-undefined]
```

1.33.13 Check that expression is not implicitly true in boolean context [truthy-bool]

Warn when the type of an expression in a boolean context does not implement `__bool__` or `__len__`. Unless one of these is implemented by a subtype, the expression will always be considered true, and there may be a bug in the condition.

As an exception, the `object` type is allowed in a boolean context. Using an iterable value in a boolean context has a separate error code (see below).

```
# 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:
     ...
```

1.33.14 Check that iterable is not implicitly true in boolean context [truthy-iterable]

Generate an error if a value of type Iterable is used as a boolean condition, since Iterable does not implement __len__ or __bool__.

Example:

```
from collections.abc import Iterable

def transform(items: Iterable[int]) -> list[int]:
    # Error: "items" has type "Iterable[int]" which can always be true in boolean␣
↪context. Consider using "Collection[int]" instead.  [truthy-iterable]
    if not items:
        return [42]
    return [x + 1 for x in items]
```

If transform is called with a Generator argument, such as int(x) for x in [], this function would not return [42] unlike what might be intended. Of course, it's possible that transform is only called with list or other container objects, and the if not items check is actually valid. If that is the case, it is recommended to annotate items as Collection[int] instead of Iterable[int].

1.33.15 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:

```
# 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
```

(continues on next page)

(continued from previous page)

```
# This line warns correctly about the typo in the attribute name
# Error: "Foo" has no attribute "nme"; maybe "name"?
f.nme = 42  # type: ignore[assignment]
```

1.33.16 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:

```
# 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())
```

You can assign the value to a temporary, otherwise unused variable to silence the error:

```
async def g() -> None:
    _ = asyncio.create_task(f())  # No error
```

1.33.17 Check that `# type: ignore` comment is used [unused-ignore]

If you use `--enable-error-code unused-ignore`, or `--warn-unused-ignores` mypy generates an error if you don't use a `# type: ignore` comment, i.e. if there is a comment, but there would be no error generated by mypy on this line anyway.

Example:

```
# Use "mypy --warn-unused-ignores ..."

def add(a: int, b: int) -> int:
    # Error: unused "type: ignore" comment
    return a + b  # type: ignore
```

Note that due to a specific nature of this comment, the only way to selectively silence it, is to include the error code explicitly. Also note that this error is not shown if the `# type: ignore` is not used due to code being statically unreachable (e.g. due to platform or version checks).

Example:

```
# Use "mypy --warn-unused-ignores ..."

import sys

try:
    # The "[unused-ignore]" is needed to get a clean mypy run
    # on both Python 3.8, and 3.9 where this module was added
```

(continues on next page)

(continued from previous page)

```
    import graphlib  # type: ignore[import,unused-ignore]
except ImportError:
    pass

if sys.version_info >= (3, 9):
    # The following will not generate an error on either
    # Python 3.8, or Python 3.9
    42 + "testing..."  # type: ignore
```

1.33.18 Check that @override is used when overriding a base class method [explicit-override]

If you use *--enable-error-code explicit-override* mypy generates an error if you override a base class method without using the @override decorator. An error will not be emitted for overrides of `__init__` or `__new__`. See PEP 698.

Note

Starting with Python 3.12, the @override decorator can be imported from typing. To use it with older Python versions, import it from typing_extensions instead.

Example:

```
# mypy: enable-error-code="explicit-override"

from typing import override

class Parent:
    def f(self, x: int) -> None:
        pass

    def g(self, y: int) -> None:
        pass


class Child(Parent):
    def f(self, x: int) -> None:  # Error: Missing @override decorator
        pass

    @override
    def g(self, y: int) -> None:
        pass
```

1.33.19 Check that overrides of mutable attributes are safe [mutable-override]

mutable-override will enable the check for unsafe overrides of mutable attributes. For historical reasons, and because this is a relatively common pattern in Python, this check is not enabled by default. The example below is unsafe, and will be flagged when this error code is enabled:

```
from typing import Any

```

(continues on next page)

(continued from previous page)

```
class C:
    x: float
    y: float
    z: float

class D(C):
    x: int  # Error: Covariant override of a mutable attribute
            # (base class "C" defined the type as "float",
            # expression has type "int")  [mutable-override]
    y: float  # OK
    z: Any  # OK

def f(c: C) -> None:
    c.x = 1.1
d = D()
f(d)
d.x >> 1  # This will crash at runtime, because d.x is now float, not an int
```

1.33.20 Check that `reveal_type` is imported from typing or typing_extensions [unimported-reveal]

Mypy used to have `reveal_type` as a special builtin that only existed during type-checking. In runtime it fails with expected `NameError`, which can cause real problem in production, hidden from mypy.

But, in Python3.11 `typing.reveal_type()` was added. `typing_extensions` ported this helper to all supported Python versions.

Now users can actually import `reveal_type` to make the runtime code safe.

> **Note**
>
> Starting with Python 3.11, the `reveal_type` function can be imported from `typing`. To use it with older Python versions, import it from `typing_extensions` instead.

```
# mypy: enable-error-code="unimported-reveal"

x = 1
reveal_type(x)  # Note: Revealed type is "builtins.int" \
                # Error: Name "reveal_type" is not defined
```

Correct usage:

```
# mypy: enable-error-code="unimported-reveal"
from typing import reveal_type   # or `typing_extensions`

x = 1
# This won't raise an error:
reveal_type(x)  # Note: Revealed type is "builtins.int"
```

When this code is enabled, using `reveal_locals` is always an error, because there's no way one can import it.

1.33.21 Check that explicit Any type annotations are not allowed [explicit-any]

If you use *--disallow-any-explicit*, mypy generates an error if you use an explicit Any type annotation.

Example:

```
# mypy: disallow-any-explicit
from typing import Any
x: Any = 1  # Error: Explicit "Any" type annotation  [explicit-any]
```

1.33.22 Check that match statements match exhaustively [exhaustive-match]

If enabled with *--enable-error-code exhaustive-match*, mypy generates an error if a match statement does not match all possible cases/types.

Example:

```
import enum


class Color(enum.Enum):
    RED = 1
    BLUE = 2

val: Color = Color.RED

# OK without --enable-error-code exhaustive-match
match val:
    case Color.RED:
        print("red")

# With --enable-error-code exhaustive-match
# Error: Match statement has unhandled case for values of type "Literal[Color.BLUE]"
match val:
    case Color.RED:
        print("red")

# OK with or without --enable-error-code exhaustive-match, since all cases are handled
match val:
    case Color.RED:
        print("red")
    case _:
        print("other")
```

1.33.23 Error if an untyped decorator makes a typed function effectively untyped [untyped-decorator]

If enabled with *--disallow-untyped-decorators* mypy generates an error if a typed function is wrapped by an untyped decorator (as this would effectively remove the benefits of typing the function).

Example:

```
def printing_decorator(func):
    def wrapper(*args, **kwds):
        print("Calling", func)
```

(continues on next page)

(continued from previous page)

```
        return func(*args, **kwds)
    return wrapper
# A decorated function.
@printing_decorator  # E: Untyped decorator makes function "add_forty_two" untyped␣
↪[untyped-decorator]
def add_forty_two(value: int) -> int:
    return value + 42
```

1.34 Additional features

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

1.34.1 Dataclasses

The `dataclasses` module allows defining and customizing simple boilerplate-free classes. They can be defined using the `@dataclasses.dataclass` decorator:

```
from dataclasses import dataclass, field

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

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

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

```
from dataclasses import dataclass

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

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

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 3.12 syntax):

```
from dataclasses import dataclass

@dataclass
class BoxedData[T]:
    data: T
    label: str
```

(continues on next page)

(continued from previous page)

```
def unbox[T](bd: BoxedData[T]) -> T:
    ...

val = unbox(BoxedData(42, "<important>"))  # OK, inferred type is int
```

For more information see official docs and **PEP 557**.

Caveats/Known Issues

Some functions in the `dataclasses` module, such as `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 example does **not** work:

```
from dataclasses import dataclass

dataclass_alias = dataclass
def dataclass_wrapper(cls):
  return dataclass(cls)

@dataclass_alias
class AliasDecorated:
  """
  Mypy doesn't recognize this as a dataclass because it is decorated by an
  alias of `dataclass` rather than by `dataclass` itself.
  """
  attribute: int

AliasDecorated(attribute=1) # error: Unexpected keyword argument
```

To have Mypy recognize a wrapper of `dataclasses.dataclass` as a dataclass decorator, consider using the `dataclass_transform()` decorator (example uses Python 3.12 syntax):

```
from dataclasses import dataclass, Field
from typing import dataclass_transform

@dataclass_transform(field_specifiers=(Field,))
def my_dataclass[T](cls: type[T]) -> type[T]:
    ...
    return dataclass(cls)
```

1.34.2 Data Class Transforms

Mypy supports the `dataclass_transform()` decorator as described in PEP 681.

> **Note**
>
> Pragmatically, mypy will assume such classes have the internal attribute `__dataclass_fields__` (even though they might lack it in runtime) and will assume functions such as `dataclasses.is_dataclass()` and `dataclasses.fields()` treat them as if they were dataclasses (even though they may fail at runtime).

1.34.3 The attrs package

attrs is a package that lets you define classes without writing boilerplate code. Mypy can detect uses of the package and will generate the necessary method definitions for decorated classes using the type annotations it finds. Type annotations can be added as follows:

```
import attrs

@attrs.define
class A:
    one: int
    two: int = 7
    three: int = attrs.field(8)
```

If you're using `auto_attribs=False` you must use `attrs.field`:

```
import attrs

@attrs.define
class A:
    one: int = attrs.field()          # Variable annotation (Python 3.6+)
    two = attrs.field()  # type: int  # Type comment
    three = attrs.field(type=int)     # type= argument
```

Typeshed has a couple of "white lie" annotations to make type checking easier. `attrs.field()` and `attrs.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 attrs

@attrs.define
class A:
    one: int = attrs.field(8)
    two: dict[str, str] = attrs.Factory(dict)
    bad: str = attrs.field(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 attrs.field()` 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 attrs
YES = True
@attrs.define(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.

1.34.4 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).

1.34.5 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`:

```
from collections.abc import Callable
from mypy_extensions import (Arg, DefaultArg, NamedArg,
                             DefaultNamedArg, VarArg, KwArg)

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

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

f: F = func
```

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

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

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

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

def DefaultArg(type=Any, name=None):
    # An optional positional argument (i.e. with a default value).
```

(continues on next page)

(continued from previous page)

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

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

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

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

is equivalent to the following:

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

A `Callable` with unspecified argument types, such as

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

is (roughly) equivalent to

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

> **Note**
>
> 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.

1.35 Frequently Asked Questions

1.35.1 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.

1.35.2 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.

1.35.3 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.

1.35.4 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.

1.35.5 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.

1.35.6 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**.

1.35.7 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.

1.35.8 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, a few Python features 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 defined 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.

1.35.9 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.

1.35.10 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).

1.35.11 Mypy is a cool project. Can I help?

Any help is much appreciated! Contact the developers if you would like to contribute. Any help related to development, design, publicity, documentation, testing, web site maintenance, financing, etc. can be helpful. You can learn a lot by contributing, and anybody can help, even beginners! However, some knowledge of compilers and/or type systems is essential if you want to work on mypy internals.

1.36 Mypy Release Notes

1.36.1 Next Release

1.36.2 Mypy 2.1

We've just uploaded mypy 2.1.0 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

librt.vecs: Fast Growable Array Type for Mypyc

The new `librt.vecs` module provides an efficient growable array type `vec` that is optimized for mypyc use. It provides fast, packed arrays with integer and floating point value types, which can be **several times faster** than `list`, and tens of times faster than `array.array` in code compiled using mypyc. It also supports nested `vec` objects and non-value-type items, such as `vec[vec[str]]`.

Refer to the documentation for the details.

Contributed by Jukka Lehtosalo.

librt.random: Fast Pseudo-Random Number Generation

The new `librt.random` module provides fast pseudo-random number generation that is optimized for code compiled using mypyc. It can be 3x to 10x faster than the stdlib `random` module in compiled code.

Refer to the documentation for the details.

Contributed by Jukka Lehtosalo (PR 21433).

Mypyc Improvements

- Make compilation order with multiple files consistent (Piotr Sawicki, PR 21419)
- Fix crash on accessing `StopAsyncIteration` (Piotr Sawicki, PR 21406)
- Fix incremental compilation with `separate` flag (Vaggelis Danias, PR 21299)

Fixes to Crashes

- Fix crash on partial type with `--allow-redefinition` and `global` declaration (Jukka Lehtosalo, PR 21428)
- Fix broken awaitable generator patching (Ivan Levkivskyi, PR 21435)

Changes to Messages

- Fix function call error message for small number of arguments (sobolevn, PR 21432)

Other Notable Fixes and Improvements

- Rely on typeshed stubs for `slice` typing (Ivan Levkivskyi, PR 21401)
- Improve negative narrowing for membership checks on tuples (Shantanu, PR 21456)
- Narrow match captures based on previous cases (Shantanu, PR 21405)
- Fix nondeterminism in overload resolution (Shantanu, PR 21455)
- Respect file config comments for stale modules (Adam Turner, PR 21444)
- Fix JSON output mode for syntax errors in parallel mode (Adam Turner, PR 21434)
- Fix type variable with values as a supertype (Ivan Levkivskyi, PR 21431)
- Add support for configuring `--num-workers` with an environment variable (Kevin Kannammalil, PR 21407)
- Respect JSON output mode for syntax errors (Adam Turner, PR 21386)
- Analyze `TypedDict` decorators (Pranav Manglik, PR 21267)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Adam Turner
- Ivan Levkivskyi
- Jukka Lehtosalo
- Kevin Kannammalil
- Piotr Sawicki
- Shantanu
- sobolevn
- Vaggelis Danias

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.3 Mypy 2.0

We've just uploaded mypy 2.0.0 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. There are also changes to options and defaults. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Enable `--local-partial-types` by Default

This flag affects the inference of types based on assignments in other scopes. For now, explicitly disabling this continues to be supported, but this support will be removed in the future as the legacy behaviour is hard to support with other current and future features in mypy, like the daemon or the new implementation of flexible redefinitions.

Contributed by Ivan Levkivskyi, Jukka Lehtosalo, Shantanu in PR 21163.

Enable `--strict-bytes` by Default

Per PEP 688, mypy no longer treats `bytearray` and `memoryview` values as assignable to the `bytes` type.

Contributed by Shantanu in PR 18371.

New Behavior for `--allow-redefinition`

The `--allow-redefinition` flag now behaves like `--allow-redefinition-new` in mypy 1.20 and earlier. The new behavior is generally more flexible. For example, you can have different types for a variable in different blocks:

```
# mypy: allow-redefinition

def foo(cond: bool) -> None:
    if cond:
        for x in ["a", "b"]:
            # Type of "x" is "str" here
            ...
    else:
        for x in [1, 2]:
            # Type of "x" is "int" here
            ...
```

The new behavior requires `--local-partial-types`, which is now enabled by default.

However, `--allow-redefinition` doesn't allow giving two type annotations for the same variable. The old behavior (sometimes) allows this. Code like this now generates an error when using `--allow-redefinition`:

```
def foo() -> None:
    x: list[int] = []
    ...
    x: list[str] = []  # Error: "x" redefined
    ...
```

You can still use `--allow-redefinition-old` to fall back to the old behavior. We have no plans to remove the legacy behavior, but the old functionality is maintained on a best effort basis.

Contributed by Jukka Lehtosalo in PR 21276.

Parallel Type Checking

Mypy now supports experimental parallel and incremental type checking. Use `--num-workers N` or `-nN` to use `N` worker processes to type check in parallel. The speedup depends on the import structure of your codebase and your environment, but for large projects we've seen performance gains of **up to 5x** when using 8 worker processes.

Parallel type checking implicitly enables the new native parser. There are still some minor semantic differences between parallel and non-parallel modes, which we will be fixing in future mypy releases.

Contributed by Ivan Levkivskyi, with additional contributions from Emma Smith and Jukka Lehtosalo.

Recent related changes since the last release:

- Freeze garbage collection in parallel workers for 4-5% speedup (Ivan Levkivskyi, PR 21302)
- Expose `--num-workers` and `--native-parser` (Ivan Levkivskyi, PR 21387)
- Split type checking into interface and implementation in parallel workers (Ivan Levkivskyi, PR 21119)
- Batch module groups for parallel processing (Ivan Levkivskyi, PR 21287)
- Optimize parallel worker startup (Ivan Levkivskyi, PR 21203)
- Parse files in parallel when possible (Ivan Levkivskyi, PR 21175)
- Use parallel parsing at all stages (Ivan Levkivskyi, PR 21266)
- Fix sequential bottleneck in parallel parsing (Jukka Lehtosalo, PR 21291)
- Fail fast when a user tries to generate reports with parallel workers (Ivan Levkivskyi, PR 21341)
- Partially support old NumPy plugin in parallel type checking (Ivan Levkivskyi, PR 21324)
- Handle reachability consistently in parallel type checking (Ivan Levkivskyi, PR 21322)
- Always respect `@no_type_check` in parallel type checking (Ivan Levkivskyi, PR 21320)
- Minor fixes in parallel checking (Ivan Levkivskyi, PR 21319)
- Fix plugin logic in parallel type checking (Ivan Levkivskyi, PR 21252)
- Fix Windows IPC race condition when using parallel checking (Jukka Lehtosalo, PR 21228)
- Report parallel worker exit status on receive failure (Jukka Lehtosalo, PR 21224)

Drop Support for Targeting Python 3.9

Mypy no longer supports type checking code with `--python-version 3.9`. Use `--python-version 3.10` or newer.

Contributed by Shantanu, Marc Mueller in PR 21243.

Remove Special Casing of Legacy Bundled Stubs

Mypy used to bundle stubs for a few packages in versions 0.812 and earlier. To navigate the transition, mypy used to report missing types for these packages even if `--ignore-missing-imports` was set. Mypy now consistently respects `--ignore-missing-imports` for all packages.

Contributed by Shantanu in PR 18372.

Prevent Assignment to None for Non-Optional Class Variables with Type Comments

Mypy used to allow assignment to None for class variables when using type comments. This was a common idiom in Python 3.5 and earlier, prior to the introduction of variable annotations. However, this was a soundness hole and has now been removed.

Contributed by Shantanu in PR 20054.

librt.strings: String and Bytes Primitives for Mypyc

In mypy 1.20, we introduced librt as a standard library for mypyc that fills in some gaps in the Python standard library and the C API. This release adds the new module `librt.strings`, which contains utilities for building string and bytes objects, and for accessing and generating binary data:

- `StringWriter` and `BytesWriter` classes allow quickly building `str` and `bytes` objects from parts.
- `read_*` and `write_*` functions provide fast reading and writing of binary-encoded data.

Refer to the documentation for the details.

Contributed by Jukka Lehtosalo.

Mypyc Improvements

- Document `librt.time` (Jukka Lehtosalo, PR 21372)
- Mark `librt.time.time()` non-experimental (Ivan Levkivskyi, PR 21310)
- Fix `librt.time` primitive now that it is no longer experimental (Ivan Levkivskyi, PR 21318)
- Fix `librt` API/ABI version checks (Jukka Lehtosalo, PR 21311)
- Generate more type methods for classes with attribute dictionaries (Piotr Sawicki, PR 21290)
- Fix reference counting for tuple items during deallocation (Shantanu, PR 21245)
- Release new instances when `__init__` raises (Shantanu, PR 21248)
- Fix `@property` getter memory leak (Vaggelis Danias, PR 21230)
- Fix semantics for walrus expression in tuple (Shantanu, PR 21249)
- Fix crash on import errors during cleanup (Shantanu, PR 21247)
- Fix reference leak in str index (Shantanu, PR 21251)
- Fix memory leak in integer true division (Shantanu, PR 21246)
- Fix reference leaks in `list.clear()/dict.clear()` (Shantanu, PR 21244)
- Resolve type aliases in function specialization (esarp, PR 21233)
- Report an error if an acyclic class inherits from non-acyclic (Piotr Sawicki, PR 21227)
- Fix `b64decode` to match new CPython behavior (Piotr Sawicki, PR 21200)

Fixes to Crashes

- Fix crash when a file does not exist during semantic analysis (Ivan Levkivskyi, PR 21379)
- Fix parallel worker crash on syntax error (Ivan Levkivskyi, PR 21202)

Changes to Messages

- Improve error messages for unexpected keyword arguments in overloaded functions (Kevin Kannammalil, PR 20592)
- Don't suggest `Foo[...]` when `Foo(arg=...)` is used in annotation (Yosof Badr, PR 21238)
- Mention what codes are actually ignored in "not covered by type: ignore comment" note (wyattscarpenter, PR 19904)
- Improve error messages when positional argument is missing (Kevin Kannammalil, PR 20591)
- Improve "name is not defined" errors with fuzzy matching (Kevin Kannammalil, PR 20693)
- Add suggestions for misspelled module imports (Kevin Kannammalil, PR 20695)

Performance Improvements

- Replace `NamedTuple` with faster regular classes in hot paths (Shantanu, PR 21326)
- Avoid calling best-match suggestions unless the message is shown (Ivan Levkivskyi, PR 21307)
- Order cases in native parser based on AST node frequency (Jukka Lehtosalo, PR 21219)

Stubtest Improvements

- Basic support for unpack kwargs (Shantanu, PR 21024)
- Fix false positive for properties with a deleter (Pranav Manglik, PR 21259)

Documentation Updates

- Rename "value restriction" to "value-constrained type variable" (Leo Ji, PR 21112)
- Clarify that invariant-by-default applies to legacy `TypeVar` syntax (Leo Ji, PR 21108)

Improvements to the Native Parser

The new native parser is still experimental.

- Make new parser consistent with the old one (Ivan Levkivskyi, PR 21377)
- Support `--package-root` with the native parser (Ivan Levkivskyi, PR 21321)
- Improve call expressions in type annotations with the native parser (Jukka Lehtosalo, PR 21300)
- Depend on `ast-serialize` by default (Jukka Lehtosalo, PR 21297)

Other Notable Fixes and Improvements

- Fix narrowing for `AbstractSet` and `Mapping` (Shantanu, PR 21352)
- Preserve gradual guarantee when narrowing `Any` union via equality (Shantanu, PR 21368)
- Make type variable upper bound narrowing symmetric (Ivan Levkivskyi, PR 21350)
- Behave consistently when type-checking a stub package directly (Ivan Levkivskyi, PR 21330)
- Add support for `Final[...]` in dataclasses (Ivan Levkivskyi, PR 21334)
- Narrow more sequence parents (Shantanu, PR 21327)
- Better narrowing for enums and other types with known equality (Shantanu, PR 21281)
- Fix pathspec error (Ivan Levkivskyi, PR 21296)
- Use sharding for the SQLite cache (Jukka Lehtosalo, PR 21292)
- Limit type inference context fallback to the walrus operator only (Ivan Levkivskyi, PR 21294)
- Support `.git/info/exclude` for `--exclude-gitignore` (RogerJinIS, PR 21286)
- Let `--allow-redefinition` widen a global in a function with `None` initialization (Jukka Lehtosalo, PR 21285)
- Delete Python 2 extra (Shantanu, PR 18374)
- No longer narrow final globals in functions (Ivan Levkivskyi, PR 21241)
- Narrow unions containing `Any` in conditional branches (Shantanu, PR 21231)
- Propagate narrowing within chained comparisons (Shantanu, PR 21160)
- Add proper lazy deserialization (Ivan Levkivskyi, PR 21198)
- Add `install_types` to options affecting cache (Brian Schubert, PR 21070)
- Narrow `Any` in conditional type checks (Shantanu, PR 21167)
- Fix exception handler target location in new parser (Ivan Levkivskyi, PR 21185)
- Improve traceback display (Shantanu, PR 21155)
- Include two more files in the sdist: `CREDITS` and the typeshed `README` (Michael R. Crusoe, PR 21131)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Brian Schubert
- Ethan Sarp
- Ivan Levkivskyi
- Jukka Lehtosalo
- Kevin Kannammalil
- Leo Ji
- Marc Mueller
- Michael R. Crusoe
- Piotr Sawicki
- Pranav Manglik
- RogerJinIS
- Shantanu
- Vaggelis Danias
- wyattscarpenter
- Yosof Badr

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.4 Mypy 1.20

We've just uploaded mypy 1.20.0 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Planned Changes to Defaults and Flags in Mypy 2.0

As a reminder, we are planning to enable `--local-partial-types` by default in mypy 2.0, which will likely be the next feature release. This will often require at least minor code changes. This option is implicitly enabled by mypy daemon, so this makes the behavior of daemon and non-daemon modes consistent.

Note that this release improves the compatibility of `--local-partial-types` significantly to make the switch easier (see below for more).

This can also be configured in a mypy configuration file (use `False` to disable):

```
local_partial_types = True
```

For more information, refer to the documentation.

We will also enable `--strict-bytes` by default in mypy 2.0. This usually requires at most minor code changes to adopt. For more information, refer to the documentation.

Finally, `--allow-redefinition-new` will be renamed to `--allow-redefinition`. If you want to continue using the older `--allow-redefinition` semantics which are less flexible (e.g. limited support for conditional redefinitions), you can switch to `--allow-redefinition-old`, which is currently supported as an alias to the legacy `--allow-redefinition` behavior. To use `--allow-redefinition` in the upcoming mypy 2.0, you can't use `--no-local-partial-types`. For more information, refer to the documentation.

Better Type Narrowing

Mypy's implementation of narrowing has been substantially reworked. Mypy will now narrow more aggressively, more consistently, and more correctly. In particular, you are likely to notice new narrowing behavior in equality expressions (==), containment expressions (`in`), match statements, and additional expressions providing type guards.

Note that mypy (and other Python type checkers) do not model the potential for various non-local operations to invalidate narrowing assumptions. This means mypy may conclude that some of your code is unreachable and avoid further checking of it. The `--warn-unreachable` flag is useful for highlighting these cases. To reset narrowing, you can insert dummy reassignments, for instance `var = var` will reset all narrowing of `var.attr`.

Future work includes better narrowing on initial assignments, more narrowing to `Literal` types, and better checking of unreachable code.

Contributed by Shantanu Jain.

- Rework narrowing logic for equality and identity (Shantanu, PR 20492)
- Refactor equality and identity narrowing for clarity (Shantanu, PR 20595)
- Treat NotImplemented as a singleton type (Shantanu, PR 20601)
- Improve narrowing logic for Enum int and str subclasses (Shantanu, PR 20609)
- Narrow types based on collection containment (Shantanu, PR 20602)
- Refactor and improve narrowing for type(x) == t checks (Shantanu, PR 20634)
- Narrow for type expr comparisons to type exprs (Shantanu, PR 20639)
- Narrowing for comparisons against `x.__class__` (Shantanu, PR 20642)
- Better narrowing with custom equality (Shantanu, PR 20643)
- Use a single pass for core narrowing logic, add comments (Shantanu, PR 20659)
- Narrowing for final type objects (Shantanu, PR 20661)
- Avoid narrowing type[T] (Shantanu, PR 20662)
- Avoid widening to Any for checks like `type(x) is type(y:  Any)` (Shantanu, PR 20663)
- Preserve some lost narrowing, cleanup (Shantanu, PR 20674)
- Fix narrowing related code for types with overloaded `__new__` (Shantanu, PR 20676)
- Fix isinstance with unions of tuples (Shantanu, PR 20677)
- Fix regression to chained containment (Shantanu, PR 20688)
- Improve else handling with custom equality (Shantanu, PR 20692)
- Better model runtime in isinstance and type checks (Shantanu, PR 20675)
- Use –warn-unreachable and –strict-equality in more tests (Shantanu, PR 20707)

- Model exact narrowing with type(x) checks (Shantanu, PR 20703)
- Short term fix for bytes narrowing (Shantanu, PR 20704)
- Preserve narrowing in unreachable code (Shantanu, PR 20710)
- Fix bug when narrowing union containing custom eq against custom eq (Shantanu, PR 20754)
- Fix narrowing for unions (Shantanu, PR 20728)
- Unsoundly narrow away from None with custom eq (Shantanu, PR 20756)
- Improve narrowing with numeric types (Shantanu, PR 20727)
- Fix narrowing with final type objects (Shantanu, PR 20743)
- Further improve match statement narrowing against unions (Shantanu, PR 20744)
- Avoid narrowing to NewType (Shantanu, PR 20766)
- Better match narrowing for irrefutable sequence patterns (Shantanu, PR 20782)
- Remove prohibit_none_typevar_overlap (Shantanu, PR 20864)
- Fix match statement narrowing reachability for tuples (Shantanu, PR 20896)
- Better handling of generics when narrowing (Shantanu, PR 20863)
- Better match narrowing for type objects (Shantanu, PR 20872)
- Narrow Callable generic return types (Shantanu, PR 20868)
- Better match narrowing for unions of type objects (Shantanu, PR 20905)
- Improve reachability in narrowing logic (Shantanu, PR 20660)
- Better match narrowing for irrefutable mapping patterns (Shantanu, PR 20906)
- Fix match statement semantic reachability (Shantanu, PR 20968)
- Add some additional narrowing test cases (Shantanu, PR 20598)
- Move tests to check-narrowing , improve them slightly (Shantanu, PR 20637)
- Add more tests for narrowing logic (Shantanu, PR 20672)
- More testing related improvements and updates (Shantanu, PR 20709)
- Add –warn-unreachable to more tests (Shantanu, PR 20977)

Drop Support for Python 3.9

Mypy no longer supports running with Python 3.9, which has reached end of life. When running mypy with Python 3.10+, it is still possible to type check code that needs to support Python 3.9 with the `--python-version 3.9` argument. Support for this will be dropped in the first half of 2026!

Contributed by Marc Mueller (PR 20156).

Mypyc Accelerated Mypy Wheels for ARM Windows and Free Threading

For best performance, mypy can be compiled to C extension modules using mypyc. This makes mypy 3-5x faster than when interpreted with pure Python. We now build and upload mypyc accelerated mypy wheels for `win_arm64` and `cp314t-...` to PyPI, making it easy for Windows users on ARM and those using the free theading builds for Python 3.14 to realise this speedup – just `pip install` the latest mypy.

Contributed by Marc Mueller (PR mypy_mypyc-wheels#106, PR mypy_mypyc-wheels#110).

Improved Compatibility for Local Partial Types

Compatibility between mypy's default behavior and the `--local-partial-types` flag is now improved. This improves compatibility between mypy daemon and non-daemon modes, since the mypy daemon requires local partial types to be enabled.

In particular, code like this now behaves consistently independent of whether local partial types are enabled or not:

```
x = None

def foo() -> None:
    global x
    x = 1

# The inferred type of 'x' is always 'int | None'.
```

Also, we are planning to turn local partial types on by default in mypy 2.0 (to be released soon), and this makes the change much less disruptive. Explicitly disabling local partial types will continue to be supported, but the support will likely be deprecated and removed eventually, as the legacy behavior is hard to support together with some important changes we are working on, in addition to being incompatible with the mypy daemon.

This feature was contributed by Ivan Levkivskyi (PR 20938).

Python 3.14 T-String Support (PEP 750)

Mypy now supports t-strings that were introduced in Python 3.14.

- Add support for Python 3.14 t-strings (PEP 750) (Neil Schemenauer and Brian Schubert, PR 20850)
- Add implicit module dependency if using t-string (Jukka Lehtosalo, PR 20900)

Experimental New Parser

If you install mypy using `pip install mypy[native-parser]` and run mypy with `--native-parser`, you can experiment with a new Python parser. It is based on the Ruff parser, and it's more efficient than the default parser. It will also enable access to all Python syntax independent of which Python version you use to run mypy. The new parser is still not feature-complete and has known issues.

Related changes:

- Add work-in-progress implementation of a new Python parser (Jukka Lehtosalo, PR 20856)
- Skip redundant analysis pass when using the native parser (Ivan Levkivskyi, PR 21015)
- Add t-string support to native parser (Ivan Levkivskyi, PR 21007)
- Handle hex bigint literals in native parser (Ivan Levkivskyi, PR 20988)
- Pass all relevant options to native parser (Ivan Levkivskyi, PR 20984)
- Support `@no_type_check` with native parser (Ivan Levkivskyi, PR 20959)
- Fix error code handling in native parser (Ivan Levkivskyi, PR 20952)
- Add `ast-serialize` as an optional dependency (Ivan Levkivskyi, PR 21028)
- Use `native-parser` instead of `native-parse` for optional dependency (Jukka Lehtosalo, PR 21115)

Performance Improvements

Mypy now uses a binary cache format (fixed-format cache) by default to speed up incremental checking. You can still use `--no-fixed-format-cache` to use the legacy JSON cache format, but we will remove the JSON cache format in a future release. Mypy includes a tool to convert individual fixed-format cache files (.ff) to the JSON format to make it possible to inspect cache contents:

```
python -m mypy.exportjson <path> ...
```

If the SQLite cache is enabled, you will first need to convert the SQLite cache into individual files using the `misc/convert-cache.py` tool available in the mypy GitHub repository. You can also disable the SQLite cache using `--no-sqlite-cache`.

The SQLite cache (`--sqlite-cache`) is now enabled by default. It improves mypy performance significantly in certain environments where slow file system operations used to be a bottleneck.

List of all performance improvements (for mypyc improvements there is a separate section below):

- Flip fixed-format cache to on by default (Ivan Levkivskyi, PR 20758)
- Enable `--sqlite-cache` by default (Shantanu, PR 21041)
- Save work on emitting ignored diagnostics (Shantanu, PR 20621)
- Skip logging and stats collection calls if they are no-ops (Jukka Lehtosalo, PR 20839)
- Speed up large incremental builds by optimizing internal state construction (Jukka Lehtosalo, PR 20838)
- Speed up suppressed dependencies options processing (Jukka Lehtosalo, PR 20806)
- Avoid path operations that need syscalls (Jukka Lehtosalo, PR 20802)
- Use faster algorithm for topological sort (Jukka Lehtosalo, PR 20790)
- Replace old topological sort (Jukka Lehtosalo, PR 20805)
- Fix quadratic performance in dependency graph loading for incremental builds (Jukka Lehtosalo, PR 20786)
- Micro-optimize transitive dependency hash calculation (Jukka Lehtosalo, PR 20798)
- Speed up options snapshot calculation (Jukka Lehtosalo, PR 20797)
- Micro-optimize read buffering, metastore `abspath`, path joining (Shantanu, PR 20810)
- Speed up type comparisons and hashing for literal types (Shantanu, PR 20423)
- Optimize overloaded signatures check (asce, PR 20378)
- Avoid unnecessary work when checking deferred functions (Ivan Levkivskyi, PR 20860)
- Improve `--allow-redefinition-new` performance for code with loops (Ivan Levkivskyi, PR 20862)
- Avoid `setattr/getattr` with fixed format cache (Ivan Levkivskyi, PR 20826)

Improvements to Allowing Redefinitions

Mypy now allows significantly more flexible variable redefinitions when using `--allow-redefinition-new`. In particular, function parameters can now be redefined with a different type:

```
# mypy: allow-redefinition-new, local-partial-types

def process(items: list[str]) -> None:
    # Reassign parameter to a completely different type.
    # Without --allow-redefinition-new, this is a type error because
```

(continues on next page)

(continued from previous page)

```
    # list[list[str]] is not compatible with list[str].
    items = [item.split() for item in items]
    ...
```

In mypy 2.0, we will update `--allow-redefinition` to mean `--allow-redefinition-new`. This release adds `--allow-redefinition-old` as an alias of `--allow-redefinition`, which can be used to continue using the old redefinition behavior in mypy 2.0 and later.

List of changes:

- Add `--allow-redefinition-old` as an alias of `--allow-redefinition` (Ivan Levkivskyi, PR 20764)
- Allow redefinitions for function arguments (Ivan Levkivskyi, PR 20853)
- Fix regression on redefinition in deferred loop (Ivan Levkivskyi, PR 20879)
- Fix loop convergence with redefinitions (Ivan Levkivskyi, PR 20865)
- Make sure new redefinition semantics only apply to inferred variables (Ivan Levkivskyi, PR 20909)
- Fix union edge case in function argument redefinition (Ivan Levkivskyi, PR 20908)
- Show an error when old and new redefinition are enabled in a file (Ivan Levkivskyi, PR 20920)
- `--allow-redefinition-new` is no longer experimental (Jukka Lehtosalo, PR 21110)
- Fix type inference for nested union types (Ivan Levkivskyi, PR 20912)
- Fix type inference regression for multiple variables in loops (Ivan Levkivskyi, PR 20892)
- Improve type inference for empty collections in conditional contexts (Ivan Levkivskyi, PR 20851)

Incremental Checking Improvements

This release includes multiple fixes to incremental type checking:

- Invalidate cache when `--enable-incomplete-feature` changes (kaushal trivedi, PR 20849)
- Add back support for `warn_unused_configs` (Ivan Levkivskyi, PR 20801)
- Recover from corrupted fixed-format cache meta file (Jukka Lehtosalo, PR 20780)
- Distinguish not found versus skipped modules (Ivan Levkivskyi, PR 20812)
- Fix undetected submodule deletion on warm run (Ivan Levkivskyi, PR 20784)
- Fix staleness on changed follow-imports options (Ivan Levkivskyi, PR 20773)
- Verify indirect dependencies reachable on incremental run (Ivan Levkivskyi, PR 20735)
- Fix indirect dependencies for protocols (Ivan Levkivskyi, PR 20752)
- Show error locations in other modules on warm runs (Ivan Levkivskyi, PR 20635)
- Don't read errors from cache on silent import (Sjoerd Job Postmus, PR 20509)
- More robust fix for re-export of `__all__` (Ivan Levkivskyi, PR 20487)

Fixes to Crashes

- Fix crash on partially typed namespace package (Ivan Levkivskyi, PR 20742)
- Fix internal error caused by the generic type alias with an unpacked list (Kai (Kazuya Ito), PR 20689)
- Fix crash when missing format character (Shantanu, PR 20524)

- Fix crash when passing literal values as type arguments to variadic generics (Aaron Wieczorek, PR 20543)
- Fix crash on circular star import in incremental mode (Ivan Levkivskyi, PR 20511)
- Fix crash with tuple unpack inside TypeVar default (Marc Mueller, PR 20456)
- Fix crash on typevar with forward reference used in other module (Ivan Levkivskyi, PR 20334)
- Fix crash on star import of redefinition (Ivan Levkivskyi, PR 20333)
- Fix crash involving Unpack-ed `TypeVarTuple` (Shantanu, PR 20323)
- Fix crashes caused by type variable defaults in-place modifications (Stanislav Terliakov, PR 20139)
- Fix crash when calling `len()` with no arguments (Jukka Lehtosalo, PR 20774)
- Fix crash when checking `async for` inside nested comprehensions (A5rocks, PR 20540)
- Fix `ParamSpec` related crash (Stanislav Terliakov, PR 20119)

Mypyc: Faster Imports on macOS

Imports in native (compiled) modules that target other native modules that are compiled together are now significantly faster on macOS, especially on the first run after a compiled package has been installed. This also speeds up the first mypy run after installation/update on macOS.

This was contributed by Jukka Lehtosalo (PR 21101).

librt: Mypyc Standard Library

Mypyc now has a dedicated standard library, `librt`, to provide basic features that are optimized for compiled code. They are faster than corresponding Python stdlib functionality. There is no plan to replace the Python stdlib, though. We'll only include a carefully selected set of features that help with common performance bottlenecks in compiled code.

Currently, we provide `librt.base64` that has optimized SIMD (Single Instruction, Multiple Data) base64 encoding and decoding functions. In future mypyc releases we are planning to add efficient data structures, string/bytes utilities, and more.

Use `python3 -m pip install librt` to make `librt` available to compiled modules. Compiled modules don't require `librt` unless they explicitly import `librt`. If you install mypy, you will also get a compatible version of `librt` as a dependency. We will keep `librt` backward compatible, so you should always be able to update to a newer version of the library.

Related changes:

- Add minimal, experimental `librt.base64` module (Jukka Lehtosalo, PR 20226)
- Use faster base64 encode implementation in `librt.base64` (Jukka Lehtosalo, PR 20237)
- Add efficient `librt.base64.b64decode` (Jukka Lehtosalo, PR 20263)
- Enable SIMD for `librt.base64` on x86-64 (Jukka Lehtosalo, PR 20244)
- Add primitive for `librt.base64.b64decode` (Jukka Lehtosalo, PR 20272)
- Add `urlsafe_b64encode` and `urlsafe_b64decode` to `librt.base64` (Jukka Lehtosalo, PR 20274)
- Make `librt.base64` non-experimental (Ivan Levkivskyi, PR 20783)
- Support pyodide for Python 3.12 (Michael R. Crusoe, PR 20342)
- Support pyodide via the NEON intrinsics (Michael R. Crusoe, PR 20316)
- Fix `librt` compilation on platforms with OpenMP (Ivan Levkivskyi, PR 20583)
- Fix cross-compiling `librt` by enabling x86_64 optimizations with pragmas (James Le Cuirot, PR 20815)

- Use existing SIMD CPU dispatch by customizing build flags (Michael R. Crusoe, PR 20253)
- Document `librt` and `librt.base64` (Jukka Lehtosalo, PR 21114)

Mypyc: Acyclic Classes

Mypyc now supports defining acyclic native classes that don't participate in the tracing garbage collection:

```
from mypy_extensions import mypyc_attr

@mypyc_attr(acyclic=True)
class Item:
    def __init__(self, key: str, value: str) -> None:
        self.key = key
        self.value = value
```

Allocating and freeing instances of acyclic classes is faster than regular native class instances, and they use less memory, but if they participate in reference cycles, there may be memory leaks.

This was contributed by Jukka Lehtosalo (PR 20795).

Additional Mypyc Fixes and Improvements

- Fix range loop variable off-by-one after loop exit (Vaggelis Danias, PR 21098)
- Fix memory leak on property setter call (Piotr Sawicki, PR 21095)
- Fix `ClassVar` self-references in class bodies (Vaggelis Danias, PR 21011)
- Fix cross-module class attribute defaults causing KeyError (Vaggelis Danias, PR 21012)
- Fix shadow vtable misalignment for `@property` getters/setters (Vaggelis Danias, PR 21010)
- Fix lambda inside comprehension (Vaggelis Danias, PR 21009)
- Use cached ASCII characters in `CPyStr_GetItem` (Vaggelis Danias, PR 21035)
- Speed up int to bytes conversion (Piotr Sawicki, PR 21036)
- Add missing primitive documentation (Jukka Lehtosalo, PR 21037)
- Fix undefined behavior in generated C (Jukka Lehtosalo, PR 21094)
- Fix vtable construction for deep trait inheritance (Vaggelis Danias, PR 20917)
- Fix `__init_subclass__` running before `ClassVar` instantiations (Vaggelis Danias, PR 20916)
- Add support for `str.lower()` and `str.upper()` (Vaggelis Danias, PR 20948)
- Add `str.isdigit()` primitive (Vaggelis Danias, PR 20893)
- Add `str.isalnum()` primitive (Vaggelis Danias, PR 20852)
- Add `str.isspace()` primitive (Vaggelis Danias, PR 20842)
- Reduce memory usage when compiling large files (Vaggelis Danias, PR 20897)
- Generate error if using Python 3.14 t-string (Jukka Lehtosalo, PR 20899)
- Fix undefined attribute in nested coroutines (Piotr Sawicki, PR 20654)
- Do not emit tracebacks with negative line numbers (Piotr Sawicki, PR 20641)
- Fix crash on multiple nested decorated functions with same name (Piotr Sawicki, PR 20666)
- Fix `Final` load in unreachable branches (BobTheBuidler, PR 20617)

- Add new primitive for `int.to_bytes` (BobTheBuidler, PR 19674)
- Support constant folding in f-string to `str` conversion (BobTheBuidler, PR 19970)
- Implement `bytes.endswith` (esarp, PR 20447)
- Fix coercion from short tagged int to fixed-width int (Jukka Lehtosalo, PR 20587)
- Fix generation of function wrappers for decorated functions (Piotr Sawicki, PR 20584)
- Generate function wrappers for each callable class instance (Piotr Sawicki, PR 20575)
- Speed up `ord(str[n])` by inlining (Jukka Lehtosalo, PR 20578)
- Add inline primitives for `bytes.__getitem__` (Jukka Lehtosalo, PR 20552)
- Add primitive type for `bytearray` (Jukka Lehtosalo, PR 20551)
- Fix and clean up `bytearray` support in primitives (Jukka Lehtosalo, PR 20550)
- Enable `--strict-bytes` by default in mypyc (and require it) (Jukka Lehtosalo, PR 20548)
- Fix exception reraising when awaiting a future (Piotr Sawicki, PR 20547)
- Add inline primitives for BytesBuilder get item and set item (Jukka Lehtosalo, PR 20546)
- Optimize loops over `enumerate`, `map`, `zip`, `range`, and other builtins with known lengths (BobTheBuidler, PR 19927)
- Raise `ValueError` if int too big for native int type (Jukka Lehtosalo, PR 20385)
- Fix generator regression with empty tuple (BobTheBuidler, PR 20371)
- Improve constant folding for `len()` of string literals and Final values (BobTheBuidler, PR 20074)
- Add primitive for `bytes.startswith` (esarp, PR 20387)
- Fix calling async methods through vectorcall (Piotr Sawicki, PR 20393)
- Extend loop optimization to use constant folding for determining sequence lengths (BobTheBuidler, PR 19930)
- Wrap async functions with function-like type (Piotr Sawicki, PR 20260)
- Add a primitive for bytes `translate` method (Jukka Lehtosalo, PR 20305)
- Add primitives for bytes and str multiply (Jukka Lehtosalo, PR 20303)
- Match int arguments to primitives with native int parameters (Jukka Lehtosalo, PR 20299)
- Allow disabling extra flags with `MYPYC_NO_EXTRA_FLAGS` environment variable (James Hilliard, PR 20507)
- Fix unsupported imports for type annotations (Lukas Geiger, PR 20390)
- Fix unaligned memory access in librt internal helper function (Gregor Riepl, PR 20474)
- Fix build issue (James Hilliard, PR 20510)

Removed Flags `--force-uppercase-builtins` and `--force-union-syntax`

The `--force-uppercase-builtins` flag was deprecated and has been a no-op since mypy 1.17.0. Since mypy has dropped support for Python 3.9, the `--force-union-syntax` flag is no longer necessary.

Contributed by Marc Mueller (PR 20410) and (PR 20405).

Stubgen Improvements

- Fix mis-parsing of double colon ("::") (Jeremy Nimmer, PR 20285)

Stubtest Improvements

- Attempt to resolve decorators from their type (Shantanu, PR 20867)
- Fix crash on instances with redefined `__class__` (sobolevn, PR 20926)
- Improve checking of positional-only parameters in dunder methods (Brian Schubert, PR 19593)
- Check `Final` variables with literal values against runtime (Vikash Kumar, PR 20858)
- Fix duplicate errors with invalid line numbers (Joren Hammudoglu, PR 20417)
- Ignore `__conditional_annotations__` (Joren Hammudoglu, PR 20392)
- Transparent `@type_check_only` types (Joren Hammudoglu, PR 20352)
- Check runtime availability of private types not marked `@type_check_only` (Brian Schubert, PR 19574)

Documentation Updates

- Document semantics of function argument redefinition (Ivan Levkivskyi, PR 20910)
- Update common issues: document how the type ignore must come first (wyattscarpenter, PR 20886)
- Update "type inference and annotations" (Kai (Kazuya Ito), PR 20619)
- Document unreachability handling of `return NotImplemented` (wyattscarpenter, PR 20561)

Changes to Messages

- Simpler/cleaner `reveal_type()` (Ivan Levkivskyi, PR 20929)
- Use "parameter" instead of "argument" in overload error (Kai (Kazuya Ito), PR 20994)
- Use "parameter" instead of "argument" in unpacking error (Kai (Kazuya Ito), PR 20683)
- Use "type arguments" instead of "type parameters" for bare generics (Kai (Kazuya Ito), PR 20494)
- Update "arguments" to "parameters" (Kai (Kazuya Ito), PR 20711)
- Use "parameter" instead of "argument" in default value error messages (Kai (Kazuya Ito), PR 20964)
- Update error message for parameter overlap in TypedDict (Kai (Kazuya Ito), PR 20956)
- Correct "Duplicate argument" error messages (Kai (Kazuya Ito), PR 20957)
- Use standard formatting for note on unexpected keyword (Ivan Levkivskyi, PR 20808)
- Fix edge cases in pretty formatting (Ivan Levkivskyi, PR 20809)
- Add function definition notes for missing named argument errors (Kevin Kannammalil, PR 20794)
- Make overloaded constructors consistent in error messages (Ivan Levkivskyi, PR 20483)
- Improve error message for invalid Python package (Shantanu, PR 20482)
- Emit end line/column in JSON format for span tracking (Adam Turner, PR 20734)
- Wrap callable in union syntax (Marc Mueller, PR 20406)

Other Notable Fixes and Improvements

- Unify handling of self attributes in daemon (Ivan Levkivskyi, PR 21025)
- Improve support for `hasattr()` (Ivan Levkivskyi, PR 20914)
- Warn when `@disjoint_base` is used on protocols or TypedDicts (Brian Schubert, PR 21029)
- Model tuple type aliases better (Shantanu, PR 20967)
- Fix incorrect type inference for container literals inside loops (Ivan Levkivskyi, PR 20875)
- Fix edge case in type comparison for type variables with narrowed bounds (Ivan Levkivskyi, PR 20874)
- Prohibit access via class for instance-only attributes (Ivan Levkivskyi, PR 20855)
- Update JSON export tool to support binary meta cache files (Jukka Lehtosalo, PR 20096)
- Better handling for uncaught top-level exceptions (Ivan Levkivskyi, PR 20749)
- Attrs field level `kw_only=False` overrides `kw_only=True` at class level (getzze, PR 20949)
- Fix `__slots__` issue with deferred base (Shantanu, PR 20573)
- Fix deferral logic for special types (Ivan Levkivskyi, PR 20678)
- Fix false negatives in walrus versus inference fallback logic (Ivan Levkivskyi, PR 20622)
- Fix enum value inference with user-defined data type mixin (E. M. Bray, PR 16320)
- Fix incorrect classification of class attributes versus enum members (Stanislav Terliakov, PR 19687)
- Improve type inference when finding common types for callable types (Shantanu, PR 18406)
- Remove special case for Union type context (Shantanu, PR 20610)
- Evaluate argument expressions in runtime evaluation order (Shantanu, PR 20491)
- Allow empty type applications for `ParamSpec` (A5rocks, PR 20572)
- Make `X[()]` count as a type application (A5rocks, PR 20568)
- Accept `value` as keyword argument in `TypeAliasType` (Ali Hamdan, PR 20556)
- Avoid treating `pass` and `...` as no-op for reachability (Shantanu, PR 20488)
- Fix specialization leak in generic `TypedDict.update()` (Aaron Wieczorek, PR 20517)
- Fix false positive redundant expression warnings with `isinstance()` on `Any | Protocol` unions (Randolf Scholz, PR 20450)
- Fix type checking of class-scope imports accessed via `self` or `cls` (bzoracler, PR 20480)
- Prevent synthetic intersections from leaking to module public interfaces (bzoracler, PR 20459)
- Fix package build failure when the compiler types are not defined (Steven Pitman, PR 20429)
- Fix `--strict-equality` for iteratively visited code (Christoph Tyralla, PR 19635)
- Allow literals as kwargs dict keys (Shantanu, PR 20416)
- Error for invalid varargs and varkwargs to `Any` call (Shantanu, PR 20324)
- Fix type inference for binary operators with tuple subclasses (Randolf Scholz, PR 19046)
- Improve type inference for ternary expressions with literals and collections (Randolf Scholz, PR 19563)
- Do not treat match value patterns as `isinstance` checks (Stanislav Terliakov, PR 20146)
- Fix noncommutative joins with bounded TypeVars (Shantanu, PR 20345)

- Fix error location reporting for `**rest` patterns in match statements (Marc Mueller, PR 20407)
- Fix matching against union of tuples (Saul Shanabrook, PR 19600)
- Allow `types.NoneType` in match cases (A5rocks, PR 20383)
- Treat `Literal["xyz"]` as iterable (Shantanu, PR 20347)
- Treat functions that return `None` as returning `None` (Shantanu, PR 20350)
- Fix str unpack type propagation, use dedicated error code (Shantanu, PR 20325)
- Unpack unions inside tuples in except handlers (asce, PR 17762)
- Ignore empty error codes from `type: ignore` (Donghoon Nam, PR 20052)
- Make `NoneType` annotation error use a new error code (wyattscarpenter, PR 20222)
- Fail with an explicit error on PyPy (Ivan Levkivskyi, PR 20384)
- Add hidden `--overwrite-union-syntax` option (Marc Mueller, PR 20332)
- Fix `possibly-undefined` false-positive with nesting (grayjk, PR 20276)
- Fix spurious `possibly-undefined` errors in for-else with `break` (Łukasz Langa, PR 19696)
- Avoid false `possibly-undefined` errors due to omitted unrequired else statements (Christoph Tyralla, PR 20149)
- Fix generator expression behavior for `reveal_type` (michaelm-openai, PR 20594)
- Fix error on instance property and init-only variable with the same name in a dataclass (Roberto Fernández Iglesias, PR 17219)
- Improve error messages when `__module__` or `__qualname__` are used as type annotations (A5rocks, PR 20288)
- Check for multiple type var tuples for PEP 695 (A5rocks, PR 20289)
- Improve interaction between `--local-partial-types` and hashability (Shantanu, PR 20719)
- Stop looking for `.gitignore` at top level of working tree (Colin Watson, PR 20775)
- Try fixing Cygwin build (Ivan Levkivskyi, PR 20830)
- Fix daemon dependencies in `diff-cache.py` tool (Jukka Lehtosalo, PR 20837)
- Support fixed-format cache in `diff-cache.py` tool (Jukka Lehtosalo, PR 20827)
- Update `convert-cache.py` tool to work with fixed-format caches (Ivan Levkivskyi, PR 20761)
- Write errors to a separate cache file (Ivan Levkivskyi, PR 21022)
- Write ignored lines to cache meta (Ivan Levkivskyi, PR 20747)
- Serialize raw errors in cache metas (Ivan Levkivskyi, PR 20372)
- Include `misc/{diff-cache,apply-cache-diff}.py` in sdist (Michael R. Crusoe, PR 21096)

Typeshed updates

Please see git log for full list of standard library typeshed stub changes.

Mypy 1.20.1

- Always disable sync in SQLite cache (Ivan Levkivskyi, PR 21184)
- Temporarily skip few base64 tests (Ivan Levkivskyi, PR 21193)
- Revert `dict.__or__` typeshed change (Ivan Levkivskyi, PR 21186)

- Fix narrowing for match case with variadic tuples (Shantanu, PR 21192)
- Avoid narrowing `type[T]` in type calls (Shantanu, PR 21174)
- Fix regression for catching empty tuple in except (Shantanu, PR 21153)
- Fix reachability for frozenset and dict view narrowing (Shantanu, PR 21151)
- Fix narrowing with chained comparison (Shantanu, PR 21150)
- Avoid narrowing to unreachable at module level (Shantanu, PR 21144)
- Allow dangerous identity comparisons to `Any` typed variables (Shantanu, PR 21142)
- `--warn-unused-config` should not be a strict flag (Ivan Levkivskyi, PR 21139)

Mypy 1.20.2

- Use WAL with SQLite cache and fix close (Shantanu, PR 21154)
- Adjust SQLite journal mode (Ivan Levkivskyi, PR 21217)
- Correctly aggregate narrowing information on parent expressions (Shantanu, PR 21206)
- Fix regression related to generic callables (Shantanu, PR 21208)
- Fix regression by avoiding widening types in some contexts (Shantanu, PR 21242)
- Fix slicing in non-strict optional mode (Shantanu, PR 21282)
- mypyc: Fix match statement semantics for "or" pattern (Shantanu, PR 21156)
- mypyc: Fix issue with module dunder attributes (Piotr Sawicki, PR 21275)
- Initial support for Python 3.15.0a8 (Marc Mueller, PR 21255)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- A5rocks
- Aaron Wieczorek
- Adam Turner
- Ali Hamdan
- asce
- BobTheBuidler
- Brent Westbrook
- Brian Schubert
- bzoracler
- Chris Burroughs
- Christoph Tyralla
- Colin Watson
- Donghoon Nam
- E. M. Bray
- Emma Smith

- Ethan Sarp
- George Ogden
- getzze
- grayjk
- Gregor Riepl
- Ivan Levkivskyi
- James Hilliard
- James Le Cuirot
- Jeremy Nimmer
- Joren Hammudoglu
- Kai (Kazuya Ito)
- kaushal trivedi
- Kevin Kannammalil
- Lukas Geiger
- Łukasz Langa
- Marc Mueller
- Michael R. Crusoe
- michaelm-openai
- Neil Schemenauer
- Piotr Sawicki
- Randolf Scholz
- Roberto Fernández Iglesias
- Saul Shanabrook
- Shantanu Jain
- Sjoerd Job Postmus
- sobolevn
- Stanislav Terliakov
- Steven Pitman
- Vaggelis Danias
- Vikash Kumar
- wyattscarpenter

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.5 Mypy 1.19

We've just uploaded mypy 1.19.0 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Python 3.9 Support Ending Soon

This is the last mypy feature release that supports Python 3.9, which reached end of life in October 2025.

Performance Improvements

- Switch to a more dynamic SCC processing logic (Ivan Levkivskyi, PR 20053)
- Speed up type aliases (Ivan Levkivskyi, PR 19810)

Fixed-Format Cache Improvements

Mypy uses a cache by default to speed up incremental runs by reusing partial results from earlier runs. Mypy 1.18 added a new binary fixed-format cache representation as an experimental feature. The feature is no longer experimental, and we are planning to enable it by default in a future mypy release (possibly 1.20), since it's faster and uses less space than the original, JSON-based cache format. Use `--fixed-format-cache` to enable the fixed-format cache.

Mypy now has an extra dependency on the `librt` PyPI package, as it's needed for cache serialization and deserialization.

Mypy ships with a tool to convert fixed-format cache files to the old JSON format. Example of how to use this:

```
$ python -m mypy.exportjson .mypy_cache/.../my_module.data.ff
```

This way existing use cases that parse JSON cache files can be supported when using the new format, though an extra conversion step is needed.

This release includes these improvements:

- Force-discard cache if cache format changed (Ivan Levkivskyi, PR 20152)
- Add tool to convert binary cache files to JSON (Jukka Lehtosalo, PR 20071)
- Use more efficient serialization format for long integers in cache files (Jukka Lehtosalo, PR 20151)
- More robust packing of floats in fixed-format cache (Ivan Levkivskyi, PR 20150)
- Use self-descriptive cache with type tags (Ivan Levkivskyi, PR 20137)
- Use fixed format for cache metas (Ivan Levkivskyi, PR 20088)
- Make metas more compact; fix indirect suppression (Ivan Levkivskyi, PR 20075)
- Use dedicated tags for most common cached instances (Ivan Levkivskyi, PR 19762)

PEP 747: Annotating Type Forms

Mypy now recognizes `TypeForm[T]` as a type and implements PEP 747. The feature is still experimental, and it's disabled by default. Use `--enable-incomplete-feature=TypeForm` to enable type forms. A type form object captures the type information provided by a runtime type expression. Example:

```
from typing_extensions import TypeForm

def trycast[T](typx: TypeForm[T], value: object) -> T | None: ...

def example(o: object) -> None:
    # 'int | str' below is an expression that represents a type.
    # Unlike type[T], TypeForm[T] can be used with all kinds of types,
    # including union types.
    x = trycast(int | str, o)
    if x is not None:
        # Type of 'x' is 'int | str' here
        ...
```

This feature was contributed by David Foster (PR 19596).

Fixes to Crashes

- Do not push partial types to the binder (Stanislav Terliakov, PR 20202)
- Fix crash on recursive tuple with Hashable (Ivan Levkivskyi, PR 20232)
- Fix crash related to decorated functions (Stanislav Terliakov, PR 20203)
- Do not abort constructing TypeAlias if only type parameters hold us back (Stanislav Terliakov, PR 20162)
- Use the fallback for `ModuleSpec` early if it can never be resolved (Stanislav Terliakov, PR 20167)
- Do not store deferred NamedTuple fields as redefinitions (Stanislav Terliakov, PR 20147)
- Discard partial types remaining after inference failure (Stanislav Terliakov, PR 20126)
- Fix an infinite recursion bug (Stanislav Terliakov, PR 20127)
- Fix IsADirectoryError for namespace packages when using –linecoverage-report (wyattscarpenter, PR 20109)
- Fix an internal error when creating cobertura output for namespace package (wyattscarpenter, PR 20112)
- Allow type parameters reusing the name missing from current module (Stanislav Terliakov, PR 20081)
- Prevent TypeGuardedType leak from narrowing declared type as part of type variable bound (Stanislav Terliakov, PR 20046)
- Fix crash on invalid unpack in base class (Ivan Levkivskyi, PR 19962)
- Traverse ParamSpec prefix where we should (Ivan Levkivskyi, PR 19800)
- Fix daemon crash related to imports (Ivan Levkivskyi, PR 20271)

Mypyc: Support for `__getattr__`, `__setattr__`, and `__delattr__`

Mypyc now has partial support for `__getattr__`, `__setattr__` and `__delattr__` methods in native classes.

Note that native attributes are not stored using `__dict__`. Setting attributes directly while bypassing `__setattr__` is possible by using `super().__setattr__(...)` or `object.__setattr__(...)`, but not via `__dict__`.

Example:

```
class Demo:
    _data: dict[str, str]

    def __init__(self) -> None:
```

(continues on next page)

(continued from previous page)

```
        # Initialize data dict without calling our __setattr__
        super().__setattr__("_data", {})

    def __setattr__(self, name: str, value: str) -> None:
        print(f"Setting {name} = {value!r}")

        if name == "_data":
            raise AttributeError("'_data' cannot be set")

        self._data[name] = value

    def __getattr__(self, name: str) -> str:
        print(f"Getting {name}")

        try:
            return self._data[name]
        except KeyError:
            raise AttributeError(name)

d = Demo()
d.x = "hello"
d.y = "world"

print(d.x)
print(d.y)
```

Related PRs:

- Generate `__setattr__` wrapper (Piotr Sawicki, PR 19937)
- Generate `__getattr__` wrapper (Piotr Sawicki, PR 19909)
- Support deleting attributes in `__setattr__` wrapper (Piotr Sawicki, PR 19997)

Miscellaneous Mypyc Improvements

- Fix `__new__` in native classes with inheritance (Piotr Sawicki, PR 20302)
- Fix crash on `super` in generator (Ivan Levkivskyi, PR 20291)
- Fix calling base class async method using `super()` (Jukka Lehtosalo, PR 20254)
- Fix async or generator methods in traits (Jukka Lehtosalo, PR 20246)
- Optimize equality check with string literals (BobTheBuidler, PR 19883)
- Fix inheritance of async defs (Jukka Lehtosalo, PR 20044)
- Reject invalid `mypyc_attr` args (BobTheBuidler, PR 19963)
- Optimize `isinstance` with tuple of primitive types (BobTheBuidler, PR 19949)
- Optimize away first index check in for loops if length > 1 (BobTheBuidler, PR 19933)
- Fix broken exception/cancellation handling in async def (Jukka Lehtosalo, PR 19951)
- Transform `object.__new__` inside `__new__` (Piotr Sawicki, PR 19866)
- Fix crash with NewType and other non-class types in incremental builds (Jukka Lehtosalo, PR 19837)

- Optimize container creation from expressions with length known at compile time (BobTheBuidler, PR 19503)
- Allow per-class free list to be used with inheritance (Jukka Lehtosalo, PR 19790)
- Fix object finalization (Marc Mueller, PR 19749)
- Allow defining a single-item free "list" for a native class (Jukka Lehtosalo, PR 19785)
- Speed up unary "not" (Jukka Lehtosalo, PR 19774)

Stubtest Improvements

- Check `_value_` for ellipsis-valued stub enum members (Stanislav Terliakov, PR 19760)
- Include function name in overload assertion messages (Joren Hammudoglu, PR 20063)
- Fix special case in analyzing function signature (iap, PR 19822)
- Improve `allowlist` docs with better example (sobolevn, PR 20007)

Documentation Updates

- Update duck type compatibility: mention strict-bytes and mypy 2.0 (wyattscarpenter, PR 20121)
- Document `--enable-incomplete-feature TypeForm` (wyattscarpenter, PR 20173)
- Change the inline TypedDict example (wyattscarpenter, PR 20172)
- Replace `List` with built-in `list` (PEP585) (Thiago J. Barbalho, PR 20000)
- Improve junit documentation (wyattscarpenter, PR 19867)

Other Notable Fixes and Improvements

- Fix annotated with function as type keyword list parameter (KarelKenens, PR 20094)
- Fix errors for raise NotImplemented (Shantanu, PR 20168)
- Don't let help formatter line-wrap URLs (Frank Dana, PR 19825)
- Do not cache fast container types inside lambdas (Stanislav Terliakov, PR 20166)
- Respect force-union-syntax flag in error hint (Marc Mueller, PR 20165)
- Fix type checking of dict type aliases (Shantanu, PR 20170)
- Use pretty callable formatting more often for callable expressions (Theodore Ando, PR 20128)
- Use dummy concrete type instead of `Any` when checking protocol variance (bzoracler, PR 20110)
- PEP 696: Fix swapping TypeVars with defaults (Randolf Scholz, PR 19449)
- Fix narrowing of class pattern with union type (Randolf Scholz, PR 19517)
- Do not emit unreachable warnings for lines that return `NotImplemented` (Christoph Tyralla, PR 20083)
- Fix matching against `typing.Callable` and `Protocol` types (Randolf Scholz, PR 19471)
- Make `--pretty` work better on multi-line issues (A5rocks, PR 20056)
- More precise return types for `TypedDict.get` (Randolf Scholz, PR 19897)
- Prevent false unreachable warnings for `@final` instances that occur when strict optional checking is disabled (Christoph Tyralla, PR 20045)
- Check class references to catch non-existent classes in match cases (A5rocks, PR 20042)
- Do not sort unused error codes in unused error codes warning (wyattscarpenter, PR 20036)

- Fix [name-defined] false positive in `class A[X, Y=X]:` case (sobolevn, PR 20021)
- Filter SyntaxWarnings during AST parsing (Marc Mueller, PR 20023)
- Make untyped decorator its own error code (wyattscarpenter, PR 19911)
- Support error codes from plugins in options (Sigve Sebastian Farstad, PR 19719)
- Allow returning Literals in `__new__` (James Hilton-Balfe, PR 15687)
- Inverse interface freshness logic (Ivan Levkivskyi, PR 19809)
- Do not report exhaustive-match after deferral (Stanislav Terliakov, PR 19804)
- Make `untyped_calls_exclude` invalidate cache (Ivan Levkivskyi, PR 19801)
- Add await to empty context hack (Stanislav Terliakov, PR 19777)
- Consider non-empty enums assignable to Self (Stanislav Terliakov, PR 19779)

Typeshed updates

Please see git log for full list of standard library typeshed stub changes.

Mypy 1.19.1

- Fix noncommutative joins with bounded TypeVars (Shantanu, PR 20345)
- Respect output format for cached runs by serializing raw errors in cache metas (Ivan Levkivskyi, PR 20372)
- Allow `types.NoneType` in match cases (A5rocks, PR 20383)
- Fix mypyc generator regression with empty tuple (BobTheBuidler, PR 20371)
- Fix crash involving Unpack-ed TypeVarTuple (Shantanu, PR 20323)
- Fix crash on star import of redefinition (Ivan Levkivskyi, PR 20333)
- Fix crash on typevar with forward ref used in other module (Ivan Levkivskyi, PR 20334)
- Fail with an explicit error on PyPy (Ivan Levkivskyi, PR 20389)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- A5rocks
- BobTheBuidler
- bzoracler
- Chainfire
- Christoph Tyralla
- David Foster
- Frank Dana
- Guo Ci
- iap
- Ivan Levkivskyi
- James Hilton-Balfe
- jhance

- Joren Hammudoglu
- Jukka Lehtosalo
- KarelKenens
- Kevin Kannammalil
- Marc Mueller
- Michael Carlstrom
- Michael J. Sullivan
- Piotr Sawicki
- Randolf Scholz
- Shantanu
- Sigve Sebastian Farstad
- sobolevn
- Stanislav Terliakov
- Stephen Morton
- Theodore Ando
- Thiago J. Barbalho
- wyattscarpenter

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.6 Mypy 1.18

We've just uploaded mypy 1.18.1 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Mypy Performance Improvements

Mypy 1.18.1 includes numerous performance improvements, resulting in about 40% speedup compared to 1.17 when type checking mypy itself. In extreme cases, the improvement can be 10x or higher. The list below is an overview of the various mypy optimizations. Many mypyc improvements (discussed in a separate section below) also improve performance.

Type caching optimizations have a small risk of causing regressions. When reporting issues with unexpected inferred types, please also check if `--disable-expression-cache` will work around the issue, as it turns off some of these optimizations.

- Improve self check performance by 1.8% (Jukka Lehtosalo, PR 19768, 19769, 19770)
- Optimize fixed-format deserialization (Ivan Levkivskyi, PR 19765)
- Use macros to optimize fixed-format deserialization (Ivan Levkivskyi, PR 19757)
- Two additional micro-optimizations (Ivan Levkivskyi, PR 19627)
- Another set of micro-optimizations (Ivan Levkivskyi, PR 19633)
- Cache common types (Ivan Levkivskyi, PR 19621)

- Skip more method bodies in third-party libraries for speed (Ivan Levkivskyi, PR 19586)
- Simplify the representation of callable types (Ivan Levkivskyi, PR 19580)
- Add cache for types of some expressions (Ivan Levkivskyi, PR 19505)
- Use cache for dictionary expressions (Ivan Levkivskyi, PR 19536)
- Use cache for binary operations (Ivan Levkivskyi, PR 19523)
- Cache types of type objects (Ivan Levkivskyi, PR 19514)
- Avoid duplicate work when checking boolean operations (Ivan Levkivskyi, PR 19515)
- Optimize generic inference passes (Ivan Levkivskyi, PR 19501)
- Speed up the default plugin (Jukka Lehtosalo, PRs 19385 and 19462)
- Remove nested imports from the default plugin (Ivan Levkivskyi, PR 19388)
- Micro-optimize type expansion (Jukka Lehtosalo, PR 19461)
- Micro-optimize type indirection (Jukka Lehtosalo, PR 19460)
- Micro-optimize the plugin framework (Jukka Lehtosalo, PR 19464)
- Avoid temporary set creation in subtype checking (Jukka Lehtosalo, PR 19463)
- Subtype checking micro-optimization (Jukka Lehtosalo, PR 19384)
- Return early where possible in subtype check (Stanislav Terliakov, PR 19400)
- Deduplicate some types before joining (Stanislav Terliakov, PR 19409)
- Speed up type checking by caching argument inference context (Jukka Lehtosalo, PR 19323)
- Optimize binding method self argument type and deprecation checks (Ivan Levkivskyi, PR 19556)
- Keep trivial instance types/aliases during expansion (Ivan Levkivskyi, PR 19543)

Fixed-Format Cache (Experimental)

Mypy now supports a new cache format used for faster incremental builds. It makes incremental builds up to twice as fast. The feature is experimental and currently only supported when using a compiled version of mypy. Use `--fixed-format-cache` to enable the new format, or `fixed_format_cache = True` in a configuration file.

We plan to enable this by default in a future mypy release, and we'll eventually deprecate and remove support for the original JSON-based format.

Unlike the JSON-based cache format, the new binary format is currently not easy to parse and inspect by mypy users. We are planning to provide a tool to convert fixed-format cache files to JSON, but details of the output JSON may be different from the current JSON format. If you rely on being able to inspect mypy cache files, we recommend creating a GitHub issue and explaining your use case, so that we can more likely provide support for it. (Using `MypyFile.read(binary_data)` to inspect cache data may be sufficient to support some use cases.)

This feature was contributed by Ivan Levkivskyi (PR 19668, 19735, 19750, 19681, 19752, 19815).

Flexible Variable Definitions: Update

Mypy 1.16.0 introduced `--allow-redefinition-new`, which allows redefining variables with different types, and inferring union types for variables from multiple assignments. The feature is now documented in the `--help` output, but the feature is still experimental.

We are planning to enable this by default in mypy 2.0, and we will also deprecate the older `--allow-redefinition` flag. Since the new behavior differs significantly from the older flag, we encourage users of `--allow-redefinition` to experiment with `--allow-redefinition-new` and create a GitHub issue if the new functionality doesn't support some important use cases.

This feature was contributed by Jukka Lehtosalo.

Inferred Type for Bare ClassVar

A ClassVar without an explicit type annotation now causes the type of the variable to be inferred from the initializer:

```
from typing import ClassVar

class Item:
    # Type of 'next_id' is now 'int' (it was 'Any')
    next_id: ClassVar = 1

    ...
```

This feature was contributed by Ivan Levkivskyi (PR 19573).

Disjoint Base Classes (@disjoint_base, PEP800)

Mypy now understands disjoint bases (PEP800): it recognizes the `@disjoint_base` decorator, and rejects class definitions that combine mutually incompatible base classes, and takes advantage of the fact that such classes cannot exist in reachability and narrowing logic.

This class definition will now generate an error:

```
# Error: Class "Bad" has incompatible disjoint bases
class Bad(str, Exception):
    ...
```

This feature was contributed by Jelle Zijlstra (PR 19678).

Miscellaneous New Mypy Features

- Add `--strict-equality-for-none` to flag non-overlapping comparisons involving None (Christoph Tyralla, PR 19718)
- Don't show import-related errors after a module-level assert such as `assert sys.platform == "linux"` that is always false (Stanislav Terliakov, PR 19347)

Improvements to Match Statements

- Add temporary named expressions for match subjects (Stanislav Terliakov, PR 18446)
- Fix unwrapping of assignment expressions in match subject (Marc Mueller, PR 19742)
- Omit errors for class patterns against object (Marc Mueller, PR 19709)
- Remove unnecessary error for certain match class patterns (Marc Mueller, PR 19708)
- Use union type for captured vars in or pattern (Marc Mueller, PR 19710)
- Prevent final reassignment inside match case (Omer Hadari, PR 19496)

Fixes to Crashes

- Fix crash with variadic tuple arguments to a generic type (Randolf Scholz, PR 19705)
- Fix crash when enable_error_code in pyproject.toml has wrong type (wyattscarpenter, PR 19494)
- Prevent crash for dataclass with PEP695 TypeVarTuple on Python3.13+ (Stanislav Terliakov, PR 19565)
- Fix crash on settable property alias (Ivan Levkivskyi, PR 19615)

Experimental Free-threading Support for Mypyc

All mypyc tests now pass on free-threading Python 3.14 release candidate builds. The performance of various micro-benchmarks scale well across multiple threads.

Free-threading support is still experimental. Note that native attribute access (get and set), list item access and certain other operations are still unsafe when there are race conditions. This will likely change in the future. You can follow the area-free-threading label in the mypyc issues tracker to follow progress.

Related PRs:

- Enable free-threading when compiling multiple modules (Jukka Lehtosalo, PR 19541)
- Fix `list.pop` on free-threaded builds (Jukka Lehtosalo, PR 19522)
- Make type objects immortal under free-threading (Jukka Lehtosalo, PR 19538)

Mypyc: Support `__new__`

Mypyc now has rudimentary support for user-defined `__new__` methods.

This feature was contributed by Piotr Sawicki (PR 19739).

Mypyc: Faster Generators and Async Functions

Generators and calls of async functions are now faster, sometimes by 2x or more.

Related PRs:

- Speed up for loops over native generators (Jukka Lehtosalo, PR 19415)
- Speed up native-to-native calls using await (Jukka Lehtosalo, PR 19398)
- Call generator helper directly in await expressions (Jukka Lehtosalo, PR 19376)
- Speed up generator allocation with per-type freelists (Jukka Lehtosalo, PR 19316)

Miscellaneous Mypyc Improvements

- Special-case certain Enum method calls for speed (Ivan Levkivskyi, PR 19634)
- Fix issues related to subclassing and undefined attribute tracking (Chainfire, PR 19787)
- Fix invalid C function signature (Jukka Lehtosalo, PR 19773)
- Speed up implicit `__ne__` (Jukka Lehtosalo, PR 19759)
- Speed up equality with optional str/bytes types (Jukka Lehtosalo, PR 19758)
- Speed up access to empty tuples (BobTheBuidler, PR 19654)
- Speed up calls with `*args` (BobTheBuidler, PRs 19623 and 19631)
- Speed up calls with `**kwargs` (BobTheBuidler, PR 19630)
- Optimize `type(x)`, `x.__class__`, and `<type>.__name__` (Jukka Lehtosalo, PR 19691, 19683)

- Specialize `bytes.decode` for common encodings (Jukka Lehtosalo, PR 19688)
- Speed up `in` operations using final fixed-length tuples (Jukka Lehtosalo, PR 19682)
- Optimize f-string building from final values (BobTheBuidler, PR 19611)
- Add dictionary set item for exact dict instances (BobTheBuidler, PR 19657)
- Cache length when iterating over immutable types (BobTheBuidler, PR 19656)
- Fix name conflict related to attributes of generator classes (Piotr Sawicki, PR 19535)
- Fix segfault from heap type objects with a static docstring (Brian Schubert, PR 19636)
- Unwrap NewType to its base type for additional optimizations (BobTheBuidler, PR 19497)
- Generate an export table only for separate compilation (Jukka Lehtosalo, PR 19521)
- Speed up `isinstance` with built-in types (Piotr Sawicki, PR 19435)
- Use native integers for some sequence indexing (Jukka Lehtosalo, PR 19426)
- Speed up `isinstance(obj, list)` (Piotr Sawicki, PR 19416)
- Report error on reserved method names (Piotr Sawicki, PR 19407)
- Speed up string equality (Jukka Lehtosalo, PR 19402)
- Raise `NameError` on undefined names (Piotr Sawicki, PR 19395)
- Use per-type freelists for nested functions (Jukka Lehtosalo, PR 19390)
- Simplify comparison of tuple elements (Piotr Sawicki, PR 19396)
- Generate introspection signatures for compiled functions (Brian Schubert, PR 19307)
- Fix undefined attribute checking special case (Jukka Lehtosalo, PR 19378)
- Fix comparison of tuples with different lengths (Piotr Sawicki, PR 19372)
- Speed up `list.clear` (Jahongir Qurbonov, PR 19344)
- Speed up `weakref.proxy` (BobTheBuidler, PR 19217)
- Speed up `weakref.ref` (BobTheBuidler, PR 19099)
- Speed up `str.count` (BobTheBuidler, PR 19264)

Stubtest Improvements

- Add temporary `--ignore-disjoint-bases` flag to ease PEP800 migration (Joren Hammudoglu, PR 19740)
- Flag redundant uses of `@disjoint_base` (Jelle Zijlstra, PR 19715)
- Improve signatures for `__init__` of C extension classes (Stephen Morton, PR 18259)
- Handle overloads with mixed positional-only parameters (Stephen Morton, PR 18287)
- Use "parameter" (not "argument") in error messages (PrinceNaroliya, PR 19707)
- Don't require `@disjoint_base` when `__slots__` imply finality (Jelle Zijlstra, PR 19701)
- Allow runtime-existing aliases of `@type_check_only` types (Brian Schubert, PR 19568)
- More detailed checking of type objects in stubtest (Stephen Morton, PR 18251)
- Support running stubtest in non-UTF8 terminals (Stanislav Terliakov, PR 19085)

Documentation Updates

- Add idlemypyextension to IDE integrations (CoolCat467, PR 18615)
- Document that `object` is often preferable to `Any` in APIs (wyattscarpenter, PR 19103)
- Include a detailed listing of flags enabled by `--strict` (wyattscarpenter, PR 19062)
- Update "common issues" (reveal_type/reveal_locals; note on orjson) (wyattscarpenter, PR 19059, 19058)

Other Notable Fixes and Improvements

- Remove deprecated `--new-type-inference` flag (the new algorithm has long been default) (Ivan Levkivskyi, PR 19570)
- Use empty context as a fallback for return expressions when outer context misleads inference (Ivan Levkivskyi, PR 19767)
- Fix forward references in type parameters of over-parameterized PEP695 aliases (Brian Schubert, PR 19725)
- Don't expand PEP695 aliases when checking node fullnames (Brian Schubert, PR 19699)
- Don't use outer context for 'or' expression inference when LHS is Any (Stanislav Terliakov, PR 19748)
- Recognize buffer protocol special methods (Brian Schubert, PR 19581)
- Support attribute access on enum members correctly (Stanislav Terliakov, PR 19422)
- Check `__slots__` assignments on self types (Stanislav Terliakov, PR 19332)
- Move self-argument checks after decorator application (Stanislav Terliakov, PR 19490)
- Infer empty list for `__slots__` and module `__all__` (Stanislav Terliakov, PR 19348)
- Use normalized tuples for fallback calculation (Stanislav Terliakov, PR 19111)
- Preserve literals when joining similar types (Stanislav Terliakov, PR 19279)
- Allow adjacent conditionally-defined overloads (Stanislav Terliakov, PR 19042)
- Check property decorators more strictly (Stanislav Terliakov, PR 19313)
- Support properties with generic setters (Ivan Levkivskyi, PR 19298)
- Generalize class/static method and property alias support (Ivan Levkivskyi, PR 19297)
- Re-widen custom properties after narrowing (Ivan Levkivskyi, PR 19296)
- Avoid erasing type objects when checking runtime cover (Shantanu, PR 19320)
- Include tuple fallback in constraints built from tuple types (Stanislav Terliakov, PR 19100)
- Somewhat better isinstance support on old-style unions (Shantanu, PR 19714)
- Improve promotions inside unions (Christoph Tyralla, PR 19245)
- Treat uninhabited types as having all attributes (Ivan Levkivskyi, PR 19300)
- Improve metaclass conflict checks (Robsdedude, PR 17682)
- Fixes to metaclass resolution algorithm (Robsdedude, PR 17713)
- PEP702 @deprecated: handle "combined" overloads (Christoph Tyralla, PR 19626)
- PEP702 @deprecated: include overloads in snapshot descriptions (Christoph Tyralla, PR 19613)
- Ignore overload implementation when checking `__OP__` / `__rOP__` compatibility (Stanislav Terliakov, PR 18502)
- Support `_value_` as a fallback for ellipsis Enum members (Stanislav Terliakov, PR 19352)

- Sort arguments in TypedDict overlap messages (Marc Mueller, PR 19666)
- Fix handling of implicit return in lambda (Stanislav Terliakov, PR 19642)
- Improve behavior of uninhabited types (Stanislav Terliakov, PR 19648)
- Fix overload diagnostics when `*args` and `**kwargs` both match (Shantanu, PR 19614)
- Further fix overload diagnostics for `*args/**kwargs` (Shantanu, PR 19619)
- Show type variable name in "Cannot infer type argument" (Brian Schubert, PR 19290)
- Fail gracefully on unsupported template strings (PEP750) (Brian Schubert, PR 19700)
- Revert colored argparse help for Python3.14 (Marc Mueller, PR 19721)
- Update stubinfo for latest typeshed (Shantanu, PR 19771)
- Fix dict assignment when an incompatible same-shape TypedDict exists (Stanislav Terliakov, PR 19592)
- Fix constructor type for subclasses of Any (Ivan Levkivskyi, PR 19295)
- Fix TypeGuard/TypeIs being forgotten in some cases (Brian Schubert, PR 19325)
- Fix TypeIs negative narrowing for unions of generics (Brian Schubert, PR 18193)
- dmypy suggest: Fix incorrect signature suggestion when a type matches a module name (Brian Schubert, PR 18937)
- dmypy suggest: Fix interaction with `__new__` (Stanislav Terliakov, PR 18966)
- dmypy suggest: Support Callable / callable Protocols in decorator unwrapping (Anthony Sottile, PR 19072)
- Fix missing error when redeclaring a type variable in a nested generic class (Brian Schubert, PR 18883)
- Fix for overloaded type object erasure (Shantanu, PR 19338)
- Fix TypeGuard with call on temporary object (Saul Shanabrook, PR 19577)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Mypy 1.18.2

- Fix crash on recursive alias (Ivan Levkivskyi, PR 19845)
- Add additional guidance for stubtest errors when runtime is `object.__init__` (Stephen Morton, PR 19733)
- Fix handling of None values in f-string expressions in mypyc (BobTheBuidler, PR 19846)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Ali Hamdan
- Anthony Sottile
- BobTheBuidler
- Brian Schubert
- Chainfire
- Charlie Denton
- Christoph Tyralla

- CoolCat467
- Daniel Hnyk
- Emily
- Emma Smith
- Ethan Sarp
- Ivan Levkivskyi
- Jahongir Qurbonov
- Jelle Zijlstra
- Joren Hammudoglu
- Jukka Lehtosalo
- Marc Mueller
- Omer Hadari
- Piotr Sawicki
- PrinceNaroliya
- Randolf Scholz
- Robsdedude
- Saul Shanabrook
- Shantanu
- Stanislav Terliakov
- Stephen Morton
- wyattscarpenter

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.7 Mypy 1.17

We've just uploaded mypy 1.17 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Optionally Check That Match Is Exhaustive

Mypy can now optionally generate an error if a match statement does not match exhaustively, without having to use `assert_never(...)`. Enable this by using `--enable-error-code exhaustive-match`.

Example:

```
# mypy: enable-error-code=exhaustive-match

import enum

class Color(enum.Enum):
```

(continues on next page)

(continued from previous page)

```
    RED = 1
    BLUE = 2

def show_color(val: Color) -> None:
    # error: Unhandled case for values of type "Literal[Color.BLUE]"
    match val:
        case Color.RED:
            print("red")
```

This feature was contributed by Donal Burns (PR 19144).

Further Improvements to Attribute Resolution

This release includes additional improvements to how attribute types and kinds are resolved. These fix many bugs and overall improve consistency.

- Handle corner case: protocol/class variable/descriptor (Ivan Levkivskyi, PR 19277)
- Fix a few inconsistencies in protocol/type object interactions (Ivan Levkivskyi, PR 19267)
- Refactor/unify access to static attributes (Ivan Levkivskyi, PR 19254)
- Remove inconsistencies in operator handling (Ivan Levkivskyi, PR 19250)
- Make protocol subtyping more consistent (Ivan Levkivskyi, PR 18943)

Fixes to Nondeterministic Type Checking

Previous mypy versions could infer different types for certain expressions across different runs (typically depending on which order certain types were processed, and this order was nondeterministic). This release includes fixes to several such issues.

- Fix nondeterministic type checking by making join with explicit Protocol and type promotion commute (Shantanu, PR 18402)
- Fix nondeterministic type checking caused by nonassociative of None joins (Shantanu, PR 19158)
- Fix nondeterministic type checking caused by nonassociativity of joins (Shantanu, PR 19147)
- Fix nondeterministic type checking by making join between `type` and TypeVar commute (Shantanu, PR 19149)

Remove Support for Targeting Python 3.8

Mypy now requires `--python-version 3.9` or greater. Support for targeting Python 3.8 is fully removed now. Since 3.8 is an unsupported version, mypy will default to the oldest supported version (currently 3.9) if you still try to target 3.8.

This change is necessary because typeshed stopped supporting Python 3.8 after it reached its End of Life in October 2024.

Contributed by Marc Mueller (PR 19157, PR 19162).

Initial Support for Python 3.14

Mypy is now tested on 3.14 and mypyc works with 3.14.0b3 and later. Binary wheels compiled with mypyc for mypy itself will be available for 3.14 some time after 3.14.0rc1 has been released.

Note that not all features are supported just yet.

Contributed by Marc Mueller (PR 19164)

Deprecated Flag: `--force-uppercase-builtins`

Mypy only supports Python 3.9+. The `--force-uppercase-builtins` flag is now deprecated as unnecessary, and a no-op. It will be removed in a future version.

Contributed by Marc Mueller (PR 19176)

Mypyc: Improvements to Generators and Async Functions

This release includes both performance improvements and bug fixes related to generators and async functions (these share many implementation details).

- Fix exception swallowing in async try/finally blocks with await (Chainfire, PR 19353)
- Fix AttributeError in async try/finally with mixed return paths (Chainfire, PR 19361)
- Make generated generator helper method internal (Jukka Lehtosalo, PR 19268)
- Free coroutine after await encounters StopIteration (Jukka Lehtosalo, PR 19231)
- Use non-tagged integer for generator label (Jukka Lehtosalo, PR 19218)
- Merge generator and environment classes in simple cases (Jukka Lehtosalo, PR 19207)

Mypyc: Partial, Unsafe Support for Free Threading

Mypyc has minimal, quite memory-unsafe support for the free threaded builds of 3.14. It is also only lightly tested. Bug reports and experience reports are welcome!

Here are some of the major limitations:

- Free threading only works when compiling a single module at a time.
- If there is concurrent access to an object while another thread is mutating the same object, it's possible to encounter segfaults and memory corruption.
- There are no efficient native primitives for thread synthronization, though the regular `threading` module can be used.
- Some workloads don't scale well to multiple threads for no clear reason.

Related PRs:

- Enable partial, unsafe support for free-threading (Jukka Lehtosalo, PR 19167)
- Fix incref/decref on free-threaded builds (Jukka Lehtosalo, PR 19127)

Other Mypyc Fixes and Improvements

- Derive .c file name from full module name if using multi_file (Jukka Lehtosalo, PR 19278)
- Support overriding the group name used in output files (Jukka Lehtosalo, PR 19272)
- Add note about using non-native class to subclass built-in types (Jukka Lehtosalo, PR 19236)
- Make some generated classes implicitly final (Jukka Lehtosalo, PR 19235)
- Don't simplify module prefixes if using separate compilation (Jukka Lehtosalo, PR 19206)

Stubgen Improvements

- Add import for `types` in `__exit__` method signature (Alexey Makridenko, PR 19120)
- Add support for including class and property docstrings (Chad Dombrova, PR 17964)
- Don't generate `Incomplete | None = None` argument annotation (Sebastian Rittau, PR 19097)

- Support several more constructs in stubgen's alias printer (Stanislav Terliakov, PR 18888)

Miscellaneous Fixes and Improvements

- Combine the revealed types of multiple iteration steps in a more robust manner (Christoph Tyralla, PR 19324)
- Improve the handling of "iteration dependent" errors and notes in finally clauses (Christoph Tyralla, PR 19270)
- Lessen dmypy suggest path limitations for Windows machines (CoolCat467, PR 19337)
- Fix type ignore comments erroneously marked as unused by dmypy (Charlie Denton, PR 15043)
- Fix misspelled `exhaustive-match` error code (johnthagen, PR 19276)
- Fix missing error context for unpacking assignment involving star expression (Brian Schubert, PR 19258)
- Fix and simplify error de-duplication (Ivan Levkivskyi, PR 19247)
- Disallow `ClassVar` in type aliases (Brian Schubert, PR 19263)
- Add script that prints list of compiled files when compiling mypy (Jukka Lehtosalo, PR 19260)
- Fix help message url for "None and Optional handling" section (Guy Wilson, PR 19252)
- Display fully qualified name of imported base classes in errors about incompatible overrides (Mikhail Golubev, PR 19115)
- Avoid false `unreachable`, `redundant-expr`, and `redundant-casts` warnings in loops more robustly and efficiently, and avoid multiple `revealed type` notes for the same line (Christoph Tyralla, PR 19118)
- Fix type extraction from `isinstance` checks (Stanislav Terliakov, PR 19223)
- Erase stray type variables in `functools.partial` (Stanislav Terliakov, PR 18954)
- Make inferring condition value recognize the whole truth table (Stanislav Terliakov, PR 18944)
- Support type aliases, `NamedTuple` and `TypedDict` in constrained TypeVar defaults (Stanislav Terliakov, PR 18884)
- Move dataclass `kw_only` fields to the end of the signature (Stanislav Terliakov, PR 19018)
- Provide a better fallback value for the `python_version` option (Marc Mueller, PR 19162)
- Avoid spurious non-overlapping equality error with metaclass with `__eq__` (Michael J. Sullivan, PR 19220)
- Narrow type variable bounds (Ivan Levkivskyi, PR 19183)
- Add classifier for Python 3.14 (Marc Mueller, PR 19199)
- Capitalize syntax error messages (Charulata, PR 19114)
- Infer constraints eagerly if actual is Any (Ivan Levkivskyi, PR 19190)
- Include walrus assignments in conditional inference (Stanislav Terliakov, PR 19038)
- Use PEP 604 syntax when converting types to strings (Marc Mueller, PR 19179)
- Use more lower-case builtin types in error messages (Marc Mueller, PR 19177)
- Fix example to use correct method of Stack (Łukasz Kwieciński, PR 19123)
- Forbid `.pop` of `Readonly` `NotRequired` TypedDict items (Stanislav Terliakov, PR 19133)
- Emit a friendlier warning on invalid exclude regex, instead of a stacktrace (wyattscarpenter, PR 19102)
- Enable ANSI color codes for dmypy client in Windows (wyattscarpenter, PR 19088)
- Extend special case for context-based type variable inference to unions in return position (Stanislav Terliakov, PR 18976)

Mypy 1.17.1

- Retain `None` as constraints bottom if no bottoms were provided (Stanislav Terliakov, PR 19485)
- Fix "ignored exception in `hasattr`" in dmypy (Stanislav Terliakov, PR 19428)
- Prevent a crash when InitVar is redefined with a method in a subclass (Stanislav Terliakov, PR 19453)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alexey Makridenko
- Brian Schubert
- Chad Dombrova
- Chainfire
- Charlie Denton
- Charulata
- Christoph Tyralla
- CoolCat467
- Donal Burns
- Guy Wilson
- Ivan Levkivskyi
- johnthagen
- Jukka Lehtosalo
- Łukasz Kwieciński
- Marc Mueller
- Michael J. Sullivan
- Mikhail Golubev
- Sebastian Rittau
- Shantanu
- Stanislav Terliakov
- wyattscarpenter

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.8 Mypy 1.16

We've just uploaded mypy 1.16 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Different Property Getter and Setter Types

Mypy now supports using different types for a property getter and setter:

```
class A:
    _value: int

    @property
    def foo(self) -> int:
        return self._value

    @foo.setter
    def foo(self, x: str | int) -> None:
        try:
            self._value = int(x)
        except ValueError:
            raise Exception(f"'{x}' is not a valid value for 'foo'")
```

This was contributed by Ivan Levkivskyi (PR 18510).

Flexible Variable Redefinitions (Experimental)

Mypy now allows unannotated variables to be freely redefined with different types when using the experimental `--allow-redefinition-new` flag. You will also need to enable `--local-partial-types`. Mypy will now infer a union type when different types are assigned to a variable:

```
# mypy: allow-redefinition-new, local-partial-types

def f(n: int, b: bool) -> int | str:
    if b:
        x = n
    else:
        x = str(n)
    # Type of 'x' is int | str here.
    return x
```

Without the new flag, mypy only supports inferring optional types (`X | None`) from multiple assignments, but now mypy can infer arbitrary union types.

An unannotated variable can now also have different types in different code locations:

```
# mypy: allow-redefinition-new, local-partial-types
...

if cond():
    for x in range(n):
        # Type of 'x' is 'int' here
        ...
else:
    for x in ['a', 'b']:
        # Type of 'x' is 'str' here
        ...
```

We are planning to turn this flag on by default in mypy 2.0, along with `--local-partial-types`. The feature is still experimental and has known issues, and the semantics may still change in the future. You may need to update or add type annotations when switching to the new behavior, but if you encounter anything unexpected, please create a GitHub issue.

This was contributed by Jukka Lehtosalo (PR 18727, PR 19153).

Stricter Type Checking with Imprecise Types

Mypy can now detect additional errors in code that uses `Any` types or has missing function annotations.

When calling `dict.get(x, None)` on an object of type `dict[str, Any]`, this now results in an optional type (in the past it was `Any`):

```
def f(d: dict[str, Any]) -> int:
    # Error: Return value has type "Any | None" but expected "int"
    return d.get("x", None)
```

Type narrowing using assignments can result in more precise types in the presence of `Any` types:

```
def foo(): ...

def bar(n: int) -> None:
    x = foo()
    # Type of 'x' is 'Any' here
    if n > 5:
        x = str(n)
        # Type of 'x' is 'str' here
```

When using `--check-untyped-defs`, unannotated overrides are now checked more strictly against superclass definitions.

Related PRs:

- Use union types instead of join in binder (Ivan Levkivskyi, PR 18538)
- Check superclass compatibility of untyped methods if `--check-untyped-defs` is set (Stanislav Terliakov, PR 18970)

Improvements to Attribute Resolution

This release includes several fixes to inconsistent resolution of attribute, method and descriptor types.

- Consolidate descriptor handling (Ivan Levkivskyi, PR 18831)
- Make multiple inheritance checking use common semantics (Ivan Levkivskyi, PR 18876)
- Make method override checking use common semantics (Ivan Levkivskyi, PR 18870)
- Fix descriptor overload selection (Ivan Levkivskyi, PR 18868)
- Handle union types when binding `self` (Ivan Levkivskyi, PR 18867)
- Make variable override checking use common semantics (Ivan Levkivskyi, PR 18847)
- Make descriptor handling behave consistently (Ivan Levkivskyi, PR 18831)

Make Implementation for Abstract Overloads Optional

The implementation can now be omitted for abstract overloaded methods, even outside stubs:

```
from abc import abstractmethod
from typing import overload

class C:
    @abstractmethod
    @overload
    def foo(self, x: int) -> int: ...

    @abstractmethod
    @overload
    def foo(self, x: str) -> str: ...

    # No implementation required for "foo"
```

This was contributed by Ivan Levkivskyi (PR 18882).

Option to Exclude Everything in .gitignore

You can now use `--exclude-gitignore` to exclude everything in a `.gitignore` file from the mypy build. This behaves similar to excluding the paths using `--exclude`. We might enable this by default in a future mypy release.

This was contributed by Ivan Levkivskyi (PR 18696).

Selectively Disable Deprecated Warnings

It's now possible to selectively disable warnings generated from `warnings.deprecated` using the `--deprecated-calls-exclude` option:

```
# mypy --enable-error-code deprecated
#      --deprecated-calls-exclude=foo.A
import foo

foo.A().func()  # OK, the deprecated warning is ignored
```

```
# file foo.py

from typing_extensions import deprecated

class A:
    @deprecated("Use A.func2 instead")
    def func(self): pass

    ...
```

Contributed by Marc Mueller (PR 18641)

Annotating Native/Non-Native Classes in Mypyc

You can now declare a class as a non-native class when compiling with mypyc. Unlike native classes, which are extension classes and have an immutable structure, non-native classes are normal Python classes at runtime and are fully dynamic. Example:

```
from mypy_extensions import mypyc_attr

```

(continues on next page)

(continued from previous page)

```
@mypyc_attr(native_class=False)
class NonNativeClass:
    ...

o = NonNativeClass()

# Ok, even if attribute "foo" not declared in class body
setattr(o, "foo", 1)
```

Classes are native by default in compiled modules, but classes that use certain features (such as most metaclasses) are implicitly non-native.

You can also explicitly declare a class as native. In this case mypyc will generate an error if it can't compile the class as a native class, instead of falling back to a non-native class:

```
from mypy_extensions import mypyc_attr
from foo import MyMeta

# Error: Unsupported metaclass for a native class
@mypyc_attr(native_class=True)
class C(metaclass=MyMeta):
    ...
```

Since native classes are significantly more efficient that non-native classes, you may want to ensure that certain classes always compiled as native classes.

This feature was contributed by Valentin Stanciu (PR 18802).

Mypyc Fixes and Improvements

- Improve documentation of native and non-native classes (Jukka Lehtosalo, PR 19154)
- Fix compilation when using Python 3.13 debug build (Valentin Stanciu, PR 19045)
- Show the reason why a class can't be a native class (Valentin Stanciu, PR 19016)
- Support await/yield while temporary values are live (Michael J. Sullivan, PR 16305)
- Fix spilling values with overlapping error values (Jukka Lehtosalo, PR 18961)
- Fix reference count of spilled register in async def (Jukka Lehtosalo, PR 18957)
- Add basic optimization for `sorted` (Marc Mueller, PR 18902)
- Fix access of class object in a type annotation (Advait Dixit, PR 18874)
- Optimize `list.__imul__` and `tuple.__mul__` (Marc Mueller, PR 18887)
- Optimize `list.__add__`, `list.__iadd__` and `tuple.__add__` (Marc Mueller, PR 18845)
- Add and implement primitive `list.copy()` (exertustfm, PR 18771)
- Optimize `builtins.repr` (Marc Mueller, PR 18844)
- Support iterating over keys/values/items of dict-bound TypeVar and ParamSpec.kwargs (Stanislav Terliakov, PR 18789)
- Add efficient primitives for `str.strip()` etc. (Advait Dixit, PR 18742)
- Document that `strip()` etc. are optimized (Jukka Lehtosalo, PR 18793)
- Fix mypyc crash with enum type aliases (Valentin Stanciu, PR 18725)

- Optimize `str.find` and `str.rfind` (Marc Mueller, PR 18709)
- Optimize `str.__contains__` (Marc Mueller, PR 18705)
- Fix order of steal/unborrow in tuple unpacking (Ivan Levkivskyi, PR 18732)
- Optimize `str.partition` and `str.rpartition` (Marc Mueller, PR 18702)
- Optimize `str.startswith` and `str.endswith` with tuple argument (Marc Mueller, PR 18678)
- Improve `str.startswith` and `str.endswith` with tuple argument (Marc Mueller, PR 18703)
- `pythoncapi_compat`: don't define Py_NULL if it is already defined (Michael R. Crusoe, PR 18699)
- Optimize `str.splitlines` (Marc Mueller, PR 18677)
- Mark `dict.setdefault` as optimized (Marc Mueller, PR 18685)
- Support `__del__` methods (Advait Dixit, PR 18519)
- Optimize `str.rsplit` (Marc Mueller, PR 18673)
- Optimize `str.removeprefix` and `str.removesuffix` (Marc Mueller, PR 18672)
- Recognize literal types in `__match_args__` (Stanislav Terliakov, PR 18636)
- Fix non extension classes with attribute annotations using forward references (Valentin Stanciu, PR 18577)
- Use lower-case generic types such as `list[t]` in documentation (Jukka Lehtosalo, PR 18576)
- Improve support for `frozenset` (Marc Mueller, PR 18571)
- Fix wheel build for cp313-win (Marc Mueller, PR 18560)
- Reduce impact of immortality (introduced in Python 3.12) on reference counting performance (Jukka Lehtosalo, PR 18459)
- Update math error messages for 3.14 (Marc Mueller, PR 18534)
- Update math error messages for 3.14 (2) (Marc Mueller, PR 18949)
- Replace deprecated `_PyLong_new` with `PyLongWriter` API (Marc Mueller, PR 18532)

Fixes to Crashes

- Traverse module ancestors when traversing reachable graph nodes during dmypy update (Stanislav Terliakov, PR 18906)
- Fix crash on multiple unpacks in a bare type application (Stanislav Terliakov, PR 18857)
- Prevent crash when enum/TypedDict call is stored as a class attribute (Stanislav Terliakov, PR 18861)
- Fix crash on multiple unpacks in a bare type application (Stanislav Terliakov, PR 18857)
- Fix crash on type inference against non-normal callables (Ivan Levkivskyi, PR 18858)
- Fix crash on decorated getter in settable property (Ivan Levkivskyi, PR 18787)
- Fix crash on callable with `*args` and suffix against Any (Ivan Levkivskyi, PR 18781)
- Fix crash on deferred supertype and setter override (Ivan Levkivskyi, PR 18649)
- Fix crashes on incorrectly detected recursive aliases (Ivan Levkivskyi, PR 18625)
- Report that `NamedTuple` and `dataclass` are incompatile instead of crashing (Christoph Tyralla, PR 18633)
- Fix mypy daemon crash (Valentin Stanciu, PR 19087)

Performance Improvements

These are specific to mypy. Mypyc-related performance improvements are discussed elsewhere.

- Speed up binding `self` in trivial cases (Ivan Levkivskyi, PR 19024)
- Small constraint solver optimization (Aaron Gokaslan, PR 18688)

Documentation Updates

- Improve documentation of `--strict` (lenayoung8, PR 18903)
- Remove a note about `from __future__ import annotations` (Ageev Maxim, PR 18915)
- Improve documentation on type narrowing (Tim Hoffmann, PR 18767)
- Fix metaclass usage example (Georg, PR 18686)
- Update documentation on `extra_checks` flag (Ivan Levkivskyi, PR 18537)

Stubgen Improvements

- Fix `TypeAlias` handling (Alexey Makridenko, PR 18960)
- Handle `arg=None` in C extension modules (Anthony Sottile, PR 18768)
- Fix valid type detection to allow pipe unions (Chad Dombrova, PR 18726)
- Include simple decorators in stub files (Marc Mueller, PR 18489)
- Support positional and keyword-only arguments in stubdoc (Paul Ganssle, PR 18762)
- Fall back to `Incomplete` if we are unable to determine the module name (Stanislav Terliakov, PR 19084)

Stubtest Improvements

- Make stubtest ignore `__slotnames__` (Nick Pope, PR 19077)
- Fix stubtest tests on 3.14 (Jelle Zijlstra, PR 19074)
- Support for `strict_bytes` in stubtest (Joren Hammudoglu, PR 19002)
- Understand override (Shantanu, PR 18815)
- Better checking of runtime arguments with dunder names (Shantanu, PR 18756)
- Ignore setattr and delattr inherited from object (Stephen Morton, PR 18325)

Miscellaneous Fixes and Improvements

- Add `--strict-bytes` to `--strict` (wyattscarpenter, PR 19049)
- Admit that Final variables are never redefined (Stanislav Terliakov, PR 19083)
- Add special support for `@django.cached_property` needed in `django-stubs` (sobolevn, PR 18959)
- Do not narrow types to `Never` with binder (Ivan Levkivskyi, PR 18972)
- Local forward references should precede global forward references (Ivan Levkivskyi, PR 19000)
- Do not cache module lookup results in incremental mode that may become invalid (Stanislav Terliakov, PR 19044)
- Only consider meta variables in ambiguous "any of" constraints (Stanislav Terliakov, PR 18986)
- Allow accessing `__init__` on final classes and when `__init__` is final (Stanislav Terliakov, PR 19035)
- Treat varargs as positional-only (A5rocks, PR 19022)

- Enable colored output for argparse help in Python 3.14 (Marc Mueller, PR 19021)
- Fix argparse for Python 3.14 (Marc Mueller, PR 19020)
- `dmypy suggest` can now suggest through contextmanager-based decorators (Anthony Sottile, PR 18948)
- Fix `__r<magic_methods>__` being used under the same `__<magic_method>__` hook (Arnav Jain, PR 18995)
- Prioritize `.pyi` from `-stubs` packages over bundled `.pyi` (Joren Hammudoglu, PR 19001)
- Fix missing subtype check case for `type[T]` (Stanislav Terliakov, PR 18975)
- Fixes to the detection of redundant casts (Anthony Sottile, PR 18588)
- Make some parse errors non-blocking (Shantanu, PR 18941)
- Fix PEP 695 type alias with a mix of type arguments (PEP 696) (Marc Mueller, PR 18919)
- Allow deeper recursion in mypy daemon, better error reporting (Carter Dodd, PR 17707)
- Fix swapped errors for frozen/non-frozen dataclass inheritance (Nazrawi Demeke, PR 18918)
- Fix incremental issue with namespace packages (Shantanu, PR 18907)
- Exclude irrelevant members when narrowing union overlapping with enum (Stanislav Terliakov, PR 18897)
- Flatten union before contracting literals when checking subtyping (Stanislav Terliakov, PR 18898)
- Do not add `kw_only` dataclass fields to `__match_args__` (sobolevn, PR 18892)
- Fix error message when returning long tuple with type mismatch (Thomas Mattone, PR 18881)
- Treat `TypedDict` (old-style) aliases as regular `TypedDicts` (Stanislav Terliakov, PR 18852)
- Warn about unused `type: ignore` comments when error code is disabled (Brian Schubert, PR 18849)
- Reject duplicate `ParamSpec.{args,kwargs}` at call site (Stanislav Terliakov, PR 18854)
- Make detection of enum members more consistent (sobolevn, PR 18675)
- Admit that `**kwargs` mapping subtypes may have no direct type parameters (Stanislav Terliakov, PR 18850)
- Don't suggest `types-setuptools` for `pkg_resources` (Shantanu, PR 18840)
- Suggest `scipy-stubs` for `scipy` as non-typeshed stub package (Joren Hammudoglu, PR 18832)
- Narrow tagged unions in match statements (Gene Parmesan Thomas, PR 18791)
- Consistently store settable property type (Ivan Levkivskyi, PR 18774)
- Do not blindly undefer on leaving function (Ivan Levkivskyi, PR 18674)
- Process superclass methods before subclass methods in semanal (Ivan Levkivskyi, PR 18723)
- Only defer top-level functions (Ivan Levkivskyi, PR 18718)
- Add one more type-checking pass (Ivan Levkivskyi, PR 18717)
- Properly account for `member` and `nonmember` in enums (sobolevn, PR 18559)
- Fix instance vs tuple subtyping edge case (Ivan Levkivskyi, PR 18664)
- Improve handling of Any/object in variadic generics (Ivan Levkivskyi, PR 18643)
- Fix handling of named tuples in class match pattern (Ivan Levkivskyi, PR 18663)
- Fix regression for user config files (Shantanu, PR 18656)
- Fix dmypy socket issue on GNU/Hurd (Mattias Ellert, PR 18630)
- Don't assume that for loop body index variable is always set (Jukka Lehtosalo, PR 18631)

- Fix overlap check for variadic generics (Ivan Levkivskyi, PR 18638)
- Improve support for `functools.partial` of overloaded callable protocol (Shantanu, PR 18639)
- Allow lambdas in `except*` clauses (Stanislav Terliakov, PR 18620)
- Fix trailing commas in many multiline string options in `pyproject.toml` (sobolevn, PR 18624)
- Allow trailing commas for `files` setting in `mypy.ini` and `setup.ini` (sobolevn, PR 18621)
- Fix "not callable" issue for `@dataclass(frozen=True)` with `Final` attr (sobolevn, PR 18572)
- Add missing TypedDict special case when checking member access (Stanislav Terliakov, PR 18604)
- Use lower case `list` and `dict` in invariance notes (Jukka Lehtosalo, PR 18594)
- Fix inference when class and instance match protocol (Ivan Levkivskyi, PR 18587)
- Remove support for `builtins.Any` (Marc Mueller, PR 18578)
- Update the overlapping check for tuples to account for NamedTuples (A5rocks, PR 18564)
- Fix `@deprecated` (PEP 702) with normal overloaded methods (Christoph Tyralla, PR 18477)
- Start propagating end columns/lines for `type-arg` errors (A5rocks, PR 18533)
- Improve handling of `type(x) is Foo` checks (Stanislav Terliakov, PR 18486)
- Suggest `typing.Literal` for exit-return error messages (Marc Mueller, PR 18541)
- Allow redefinitions in except/else/finally (Stanislav Terliakov, PR 18515)
- Disallow setting Python version using inline config (Shantanu, PR 18497)
- Improve type inference in tuple multiplication plugin (Shantanu, PR 18521)
- Add missing line number to `yield from` with wrong type (Stanislav Terliakov, PR 18518)
- Hint at argument names when formatting callables with compatible return types in error messages (Stanislav Terliakov, PR 18495)
- Add better naming and improve compatibility for ad hoc intersections of instances (Christoph Tyralla, PR 18506)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- A5rocks
- Aaron Gokaslan
- Advait Dixit
- Ageev Maxim
- Alexey Makridenko
- Ali Hamdan
- Anthony Sottile
- Arnav Jain
- Brian Schubert
- bzoracler
- Carter Dodd
- Chad Dombrova

- Christoph Tyralla
- Dimitri Papadopoulos Orfanos
- Emma Smith
- exertustfm
- Gene Parmesan Thomas
- Georg
- Ivan Levkivskyi
- Jared Hance
- Jelle Zijlstra
- Joren Hammudoglu
- lenayoung8
- Marc Mueller
- Mattias Ellert
- Michael J. Sullivan
- Michael R. Crusoe
- Nazrawi Demeke
- Nick Pope
- Paul Ganssle
- Shantanu
- sobolevn
- Stanislav Terliakov
- Stephen Morton
- Thomas Mattone
- Tim Hoffmann
- Tim Ruffing
- Valentin Stanciu
- Wesley Collin Wright
- wyattscarpenter

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.9 Mypy 1.15

We've just uploaded mypy 1.15 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Performance Improvements

Mypy is up to 40% faster in some use cases. This improvement comes largely from tuning the performance of the garbage collector. Additionally, the release includes several micro-optimizations that may be impactful for large projects.

Contributed by Jukka Lehtosalo

- PR 18306
- PR 18302
- PR 18298
- PR 18299

Mypyc Accelerated Mypy Wheels for ARM Linux

For best performance, mypy can be compiled to C extension modules using mypyc. This makes mypy 3-5x faster than when interpreted with pure Python. We now build and upload mypyc accelerated mypy wheels for `manylinux_aarch64` to PyPI, making it easy for Linux users on ARM platforms to realise this speedup – just `pip install` the latest mypy.

Contributed by Christian Bundy and Marc Mueller (PR mypy_mypyc-wheels#76, PR mypy_mypyc-wheels#89).

`--strict-bytes`

By default, mypy treats `bytearray` and `memoryview` values as assignable to the `bytes` type, for historical reasons. Use the `--strict-bytes` flag to disable this behavior. PEP 688 specified the removal of this special case. The flag will be enabled by default in **mypy 2.0**.

Contributed by Ali Hamdan (PR 18263) and Shantanu Jain (PR 13952).

Improvements to Reachability Analysis and Partial Type Handling in Loops

This change results in mypy better modelling control flow within loops and hence detecting several previously ignored issues. In some cases, this change may require additional explicit variable annotations.

Contributed by Christoph Tyralla (PR 18180, PR 18433).

(Speaking of partial types, remember that we plan to enable `--local-partial-types` by default in **mypy 2.0**.)

Better Discovery of Configuration Files

Mypy will now walk up the filesystem (up until a repository or file system root) to discover configuration files. See the mypy configuration file documentation for more details.

Contributed by Mikhail Shiryaev and Shantanu Jain (PR 16965, PR 18482)

Better Line Numbers for Decorators and Slice Expressions

Mypy now uses more correct line numbers for decorators and slice expressions. In some cases, you may have to change the location of a `# type: ignore` comment.

Contributed by Shantanu Jain (PR 18392, PR 18397).

Drop Support for Python 3.8

Mypy no longer supports running with Python 3.8, which has reached end-of-life. When running mypy with Python 3.9+, it is still possible to type check code that needs to support Python 3.8 with the `--python-version 3.8` argument. Support for this will be dropped in the first half of 2025!

Contributed by Marc Mueller (PR 17492).

Mypyc Improvements

- Fix `__init__` for classes with `@attr.s(slots=True)` (Advait Dixit, PR 18447)
- Report error for nested class instead of crashing (Valentin Stanciu, PR 18460)
- Fix `InitVar` for dataclasses (Advait Dixit, PR 18319)
- Remove unnecessary mypyc files from wheels (Marc Mueller, PR 18416)
- Fix issues with relative imports (Advait Dixit, PR 18286)
- Add faster primitive for some list get item operations (Jukka Lehtosalo, PR 18136)
- Fix iteration over `NamedTuple` objects (Advait Dixit, PR 18254)
- Mark mypyc package with `py.typed` (bzoracler, PR 18253)
- Fix list index while checking for `Enum` class (Advait Dixit, PR 18426)

Stubgen Improvements

- Improve dataclass init signatures (Marc Mueller, PR 18430)
- Preserve `dataclass_transform` decorator (Marc Mueller, PR 18418)
- Fix `UnpackType` for 3.11+ (Marc Mueller, PR 18421)
- Improve `self` annotations (Marc Mueller, PR 18420)
- Print `InspectError` traceback in stubgen `walk_packages` when verbose is specified (Gareth, PR 18224)

Stubtest Improvements

- Fix crash with numpy array default values (Ali Hamdan, PR 18353)
- Distinguish metaclass attributes from class attributes (Stephen Morton, PR 18314)

Fixes to Crashes

- Prevent crash with `Unpack` of a fixed tuple in PEP695 type alias (Stanislav Terliakov, PR 18451)
- Fix crash with `--cache-fine-grained --cache-dir=/dev/null` (Shantanu, PR 18457)
- Prevent crashing when `match` arms use name of existing callable (Stanislav Terliakov, PR 18449)
- Gracefully handle encoding errors when writing to stdout (Brian Schubert, PR 18292)
- Prevent crash on generic NamedTuple with unresolved typevar bound (Stanislav Terliakov, PR 18585)

Documentation Updates

- Add inline tabs to documentation (Marc Mueller, PR 18262)
- Document any `TYPE_CHECKING` name works (Shantanu, PR 18443)
- Update documentation to not mention 3.8 where possible (sobolevn, PR 18455)
- Mention `ignore_errors` in exclude documentation (Shantanu, PR 18412)
- Add `Self` misuse to common issues (Shantanu, PR 18261)

Other Notable Fixes and Improvements

- Fix literal context for ternary expressions (Ivan Levkivskyi, PR 18545)
- Ignore `dataclass.__replace__` LSP violations (Marc Mueller, PR 18464)
- Bind `self` to the class being defined when checking multiple inheritance (Stanislav Terliakov, PR 18465)
- Fix attribute type resolution with multiple inheritance (Stanislav Terliakov, PR 18415)
- Improve security of our GitHub Actions (sobolevn, PR 18413)
- Unwrap `type[Union[...]]` when solving type variable constraints (Stanislav Terliakov, PR 18266)
- Allow `Any` to match sequence patterns in match/case (Stanislav Terliakov, PR 18448)
- Fix parent generics mapping when overriding generic attribute with property (Stanislav Terliakov, PR 18441)
- Add dedicated error code for explicit `Any` (Shantanu, PR 18398)
- Reject invalid `ParamSpec` locations (Stanislav Terliakov, PR 18278)
- Stop suggesting stubs that have been removed from typeshed (Shantanu, PR 18373)
- Allow inverting `--local-partial-types` (Shantanu, PR 18377)
- Allow to use `Final` and `ClassVar` after Python 3.13 (, PR 18358)
- Update suggestions to include latest stubs in typeshed (Shantanu, PR 18366)
- Fix `--install-types` masking failure details (wyattscarpenter, PR 17485)
- Reject promotions when checking against protocols (Christoph Tyralla, PR 18360)
- Don't erase type object arguments in diagnostics (Shantanu, PR 18352)
- Clarify status in `dmypy status` output (Kcornw, PR 18331)
- Disallow no-argument generic aliases when using PEP 613 explicit aliases (Brian Schubert, PR 18173)
- Suppress errors for unreachable branches in conditional expressions (Brian Schubert, PR 18295)
- Do not allow `ClassVar` and `Final` in `TypedDict` and `NamedTuple` (sobolevn, PR 18281)
- Report error if not enough or too many types provided to `TypeAliasType` (bzoracler, PR 18308)
- Use more precise context for `TypedDict` plugin errors (Brian Schubert, PR 18293)
- Use more precise context for invalid type argument errors (Brian Schubert, PR 18290)
- Do not allow `type[]` to contain `Literal` types (sobolevn, PR 18276)
- Allow bytearray/bytes comparisons with `--strict-bytes` (Jukka Lehtosalo, PR 18255)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Advait Dixit
- Ali Hamdan
- Brian Schubert
- bzoracler
- Cameron Matsui
- Christoph Tyralla
- Gareth

- Ivan Levkivskyi
- Jukka Lehtosalo
- Kcornw
- Marc Mueller
- Mikhail f. Shiryaev
- Shantanu
- sobolevn
- Stanislav Terliakov
- Stephen Morton
- Valentin Stanciu
- Viktor Szépe
- wyattscarpenter
-

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.10 Mypy 1.14

We've just uploaded mypy 1.14 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Change to Enum Membership Semantics

As per the updated typing specification for enums, enum members must be left unannotated.

```
class Pet(Enum):
    CAT = 1  # Member attribute
    DOG = 2  # Member attribute

    # New error: Enum members must be left unannotated
    WOLF: int = 3

    species: str  # Considered a non-member attribute
```

In particular, the specification change can result in issues in type stubs (`.pyi` files), since historically it was common to leave the value absent:

```
# In a type stub (.pyi file)

class Pet(Enum):
    # Change in semantics: previously considered members,
    # now non-member attributes
    CAT: int
    DOG: int
```

(continues on next page)

(continued from previous page)

```
    # Mypy will now issue a warning if it detects this
    # situation in type stubs:
    # > Detected enum "Pet" in a type stub with zero
    # > members. There is a chance this is due to a recent
    # > change in the semantics of enum membership. If so,
    # > use `member = value` to mark an enum member,
    # > instead of `member: type`

class Pet(Enum):
    # As per the specification, you should now do one of
    # the following:
    DOG = 1  # Member attribute with value 1 and known type
    WOLF = cast(int, ...)  # Member attribute with unknown
                           # value but known type
    LION = ...  # Member attribute with unknown value and
                # # unknown type
```

Contributed by Terence Honles (PR 17207) and Shantanu Jain (PR 18068).

Support for @deprecated Decorator (PEP 702)

Mypy can now issue errors or notes when code imports a deprecated feature explicitly with a `from mod import depr` statement, or uses a deprecated feature imported otherwise or defined locally. Features are considered deprecated when decorated with `warnings.deprecated`, as specified in PEP 702.

You can enable the error code via `--enable-error-code=deprecated` on the mypy command line or `enable_error_code = deprecated` in the mypy config file. Use the command line flag `--report-deprecated-as-note` or config file option `report_deprecated_as_note=True` to turn all such errors into notes.

Deprecation errors will be enabled by default in a future mypy version.

This feature was contributed by Christoph Tyralla.

List of changes:

- Add basic support for PEP 702 (`@deprecated`) (Christoph Tyralla, PR 17476)
- Support descriptors with `@deprecated` (Christoph Tyralla, PR 18090)
- Make "deprecated" note an error, disabled by default (Valentin Stanciu, PR 18192)
- Consider all possible type positions with `@deprecated` (Christoph Tyralla, PR 17926)
- Improve the handling of explicit type annotations in assignment statements with `@deprecated` (Christoph Tyralla, PR 17899)

Optionally Analyzing Untyped Modules

Mypy normally doesn't analyze imports from third-party modules (installed using pip, for example) if there are no stubs or a py.typed marker file. To force mypy to analyze these imports, you can now use the `--follow-untyped-imports` flag or set the `follow_untyped_imports` config file option to True. This can be set either in the global section of your mypy config file, or individually on a per-module basis.

This feature was contributed by Jannick Kremer.

List of changes:

- Implement flag to allow type checking of untyped modules (Jannick Kremer, PR 17712)

- Warn about `--follow-untyped-imports` (Shantanu, PR 18249)

Support New Style Type Variable Defaults (PEP 696)

Mypy now supports type variable defaults using the new syntax described in PEP 696, which was introduced in Python 3.13. Example:

```
@dataclass
class Box[T = int]:  # Set default for "T"
    value: T | None = None

reveal_type(Box())                      # type is Box[int], since it's the default
reveal_type(Box(value="Hello World!"))  # type is Box[str]
```

This feature was contributed by Marc Mueller (PR 17985).

Improved For Loop Index Variable Type Narrowing

Mypy now preserves the literal type of for loop index variables, to support `TypedDict` lookups. Example:

```
from typing import TypedDict

class X(TypedDict):
    hourly: int
    daily: int

def func(x: X) -> int:
    s = 0
    for var in ("hourly", "daily"):
        # "Union[Literal['hourly']?, Literal['daily']?]"
        reveal_type(var)

        # x[var] no longer triggers a literal-required error
        s += x[var]
    return s
```

This was contributed by Marc Mueller (PR 18014).

Mypyc Improvements

- Document optimized bytes operations and additional str operations (Jukka Lehtosalo, PR 18242)
- Add primitives and specialization for `ord()` (Jukka Lehtosalo, PR 18240)
- Optimize `str.encode` with specializations for common used encodings (Valentin Stanciu, PR 18232)
- Fix fall back to generic operation for staticmethod and classmethod (Advait Dixit, PR 18228)
- Support unicode surrogates in string literals (Jukka Lehtosalo, PR 18209)
- Fix index variable in for loop with `builtins.enumerate` (Advait Dixit, PR 18202)
- Fix check for enum classes (Advait Dixit, PR 18178)
- Fix loading type from imported modules (Advait Dixit, PR 18158)
- Fix initializers of final attributes in class body (Jared Hance, PR 18031)
- Fix name generation for modules with similar full names (aatle, PR 18001)

- Fix relative imports in `__init__.py` (Shantanu, PR 17979)
- Optimize dunder methods (jairov4, PR 17934)
- Replace deprecated `_PyDict_GetItemStringWithError` (Marc Mueller, PR 17930)
- Fix wheel build for cp313-win (Marc Mueller, PR 17941)
- Use public PyGen_GetCode instead of vendored implementation (Marc Mueller, PR 17931)
- Optimize calls to final classes (jairov4, PR 17886)
- Support ellipsis (`...`) expressions in class bodies (Newbyte, PR 17923)
- Sync `pythoncapi_compat.h` (Marc Mueller, PR 17929)
- Add `runtests.py mypyc-fast` for running fast mypyc tests (Jukka Lehtosalo, PR 17906)

Stubgen Improvements

- Do not include mypy generated symbols (Ali Hamdan, PR 18137)
- Fix `FunctionContext.fullname` for nested classes (Chad Dombrova, PR 17963)
- Add flagfile support (Ruslan Sayfutdinov, PR 18061)
- Add support for PEP 695 and PEP 696 syntax (Ali Hamdan, PR 18054)

Stubtest Improvements

- Allow the use of `--show-traceback` and `--pdb` with stubtest (Stephen Morton, PR 18037)
- Verify `__all__` exists in stub (Sebastian Rittau, PR 18005)
- Stop telling people to use double underscores (Jelle Zijlstra, PR 17897)

Documentation Updates

- Update config file documentation (sobolevn, PR 18103)
- Improve contributor documentation for Windows (ag-tafe, PR 18097)
- Correct note about `--disallow-any-generics` flag in documentation (Abel Sen, PR 18055)
- Further caution against `--follow-imports=skip` (Shantanu, PR 18048)
- Fix the edit page button link in documentation (Kanishk Pachauri, PR 17933)

Other Notables Fixes and Improvements

- Allow enum members to have type objects as values (Jukka Lehtosalo, PR 19160)
- Show `Protocol` `__call__` for arguments with incompatible types (MechanicalConstruct, PR 18214)
- Make join and meet symmetric with `strict_optional` (MechanicalConstruct, PR 18227)
- Preserve block unreachablility when checking function definitions with constrained TypeVars (Brian Schubert, PR 18217)
- Do not include non-init fields in the synthesized `__replace__` method for dataclasses (Victorien, PR 18221)
- Disallow `TypeVar` constraints parameterized by type variables (Brian Schubert, PR 18186)
- Always complain about invalid varargs and varkwargs (Shantanu, PR 18207)
- Set default strict_optional state to True (Shantanu, PR 18198)
- Preserve type variable default None in type alias (Sukhorosov Aleksey, PR 18197)

- Add checks for invalid usage of continue/break/return in `except*` block (coldwolverine, PR 18132)
- Do not consider bare TypeVar not overlapping with None for reachability analysis (Stanislav Terliakov, PR 18138)
- Special case `types.DynamicClassAttribute` as property-like (Stephen Morton, PR 18150)
- Disallow bare `ParamSpec` in type aliases (Brian Schubert, PR 18174)
- Move long_description metadata to pyproject.toml (Marc Mueller, PR 18172)
- Support ==-based narrowing of Optional (Christoph Tyralla, PR 18163)
- Allow TypedDict assignment of Required item to NotRequired ReadOnly item (Brian Schubert, PR 18164)
- Allow nesting of Annotated with TypedDict special forms inside TypedDicts (Brian Schubert, PR 18165)
- Infer generic type arguments for slice expressions (Brian Schubert, PR 18160)
- Fix checking of match sequence pattern against bounded type variables (Brian Schubert, PR 18091)
- Fix incorrect truthyness for Enum types and literals (David Salvisberg, PR 17337)
- Move static project metadata to pyproject.toml (Marc Mueller, PR 18146)
- Fallback to stdlib json if integer exceeds 64-bit range (q0w, PR 18148)
- Fix 'or' pattern structural matching exhaustiveness (yihong, PR 18119)
- Fix type inference of positional parameter in class pattern involving builtin subtype (Brian Schubert, PR 18141)
- Fix `[override]` error with no line number when argument node has no line number (Brian Schubert, PR 18122)
- Fix some dmypy crashes (Ivan Levkivskyi, PR 18098)
- Fix subtyping between instance type and overloaded (Shantanu, PR 18102)
- Clean up new_semantic_analyzer config (Shantanu, PR 18071)
- Issue warning for enum with no members in stub (Shantanu, PR 18068)
- Fix enum attributes are not members (Terence Honles, PR 17207)
- Fix crash when checking slice expression with step 0 in tuple index (Brian Schubert, PR 18063)
- Allow union-with-callable attributes to be overridden by methods (Brian Schubert, PR 18018)
- Emit `[mutable-override]` for covariant override of attribute with method (Brian Schubert, PR 18058)
- Support ParamSpec mapping with `functools.partial` (Stanislav Terliakov, PR 17355)
- Fix approved stub ignore, remove normpath (Shantanu, PR 18045)
- Make `disallow-any-unimported` flag invertible (Séamus Ó Ceanainn, PR 18030)
- Filter to possible package paths before trying to resolve a module (falsedrow, PR 18038)
- Fix overlap check for ParamSpec types (Jukka Lehtosalo, PR 18040)
- Do not prioritize ParamSpec signatures during overload resolution (Stanislav Terliakov, PR 18033)
- Fix ternary union for literals (Ivan Levkivskyi, PR 18023)
- Fix compatibility checks for conditional function definitions using decorators (Brian Schubert, PR 18020)
- TypeGuard should be bool not Any when matching TypeVar (Evgeniy Slobodkin, PR 17145)
- Fix convert-cache tool (Shantanu, PR 17974)
- Fix generator comprehension with mypyc (Shantanu, PR 17969)
- Fix crash issue when using shadowfile with pretty (Max Chang, PR 17894)

- Fix multiple nested classes with new generics syntax (Max Chang, PR 17820)
- Better error for `mypy -p package` without py.typed (Joe Gordon, PR 17908)
- Emit error for `raise NotImplemented` (Brian Schubert, PR 17890)
- Add `is_lvalue` attribute to AttributeContext (Brian Schubert, PR 17881)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- aatle
- Abel Sen
- Advait Dixit
- ag-tafe
- Alex Waygood
- Ali Hamdan
- Brian Schubert
- Carlton Gibson
- Chad Dombrova
- Chelsea Durazo
- chiri
- Christoph Tyralla
- coldwolverine
- David Salvisberg
- Ekin Dursun
- Evgeniy Slobodkin
- falsedrow
- Gaurav Giri
- Ihor
- Ivan Levkivskyi
- jairov4
- Jannick Kremer
- Jared Hance
- Jelle Zijlstra
- jianghuyiyuan
- Joe Gordon
- John Doknjas
- Jukka Lehtosalo
- Kanishk Pachauri
- Marc Mueller

- Max Chang
- MechanicalConstruct
- Newbyte
- q0w
- Ruslan Sayfutdinov
- Sebastian Rittau
- Shantanu
- sobolevn
- Stanislav Terliakov
- Stephen Morton
- Sukhorosov Aleksey
- Séamus Ó Ceanainn
- Terence Honles
- Valentin Stanciu
- vasiliy
- Victorien
- yihong

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.11 Mypy 1.13

We've just uploaded mypy 1.13 to the Python Package Index (PyPI). Mypy is a static type checker for Python. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Note that unlike typical releases, Mypy 1.13 does not have any changes to type checking semantics from 1.12.1.

Improved Performance

Mypy 1.13 contains several performance improvements. Users can expect mypy to be 5-20% faster. In environments with long search paths (such as environments using many editable installs), mypy can be significantly faster, e.g. 2.2x faster in the use case targeted by these improvements.

Mypy 1.13 allows use of the `orjson` library for handling the cache instead of the stdlib `json`, for improved performance. You can ensure the presence of `orjson` using the `faster-cache` extra:

```
python3 -m pip install -U mypy[faster-cache]
```

Mypy may depend on `orjson` by default in the future.

These improvements were contributed by Shantanu.

List of changes:

- Significantly speed up file handling error paths (Shantanu, PR 17920)

- Use fast path in modulefinder more often (Shantanu, PR 17950)
- Let mypyc optimise os.path.join (Shantanu, PR 17949)
- Make is_sub_path faster (Shantanu, PR 17962)
- Speed up stubs suggestions (Shantanu, PR 17965)
- Use sha1 for hashing (Shantanu, PR 17953)
- Use orjson instead of json, when available (Shantanu, PR 17955)
- Add faster-cache extra, test in CI (Shantanu, PR 17978)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Shantanu Jain
- Jukka Lehtosalo

1.36.12 Mypy 1.12

We've just uploaded mypy 1.12 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Support Python 3.12 Syntax for Generics (PEP 695)

Support for the new type parameter syntax introduced in Python 3.12 is now enabled by default, documented, and no longer experimental. It was available through a feature flag in mypy 1.11 as an experimental feature.

This example demonstrates the new syntax:

```
# Generic function
def f[T](x: T) -> T: ...

reveal_type(f(1))  # Revealed type is 'int'

# Generic class
class C[T]:
    def __init__(self, x: T) -> None:
       self.x = x

c = C('a')
reveal_type(c.x)  # Revealed type is 'str'

# Type alias
type A[T] = C[list[T]]
```

For more information, refer to the documentation.

These improvements are included:

- Document Python 3.12 type parameter syntax (Jukka Lehtosalo, PR 17816)
- Further documentation updates (Jukka Lehtosalo, PR 17826)

- Allow Self return types with contravariance (Jukka Lehtosalo, PR 17786)
- Enable new type parameter syntax by default (Jukka Lehtosalo, PR 17798)
- Generate error if new-style type alias used as base class (Jukka Lehtosalo, PR 17789)
- Inherit variance if base class has explicit variance (Jukka Lehtosalo, PR 17787)
- Fix crash on invalid type var reference (Jukka Lehtosalo, PR 17788)
- Fix covariance of frozen dataclasses (Jukka Lehtosalo, PR 17783)
- Allow covariance with attribute that has "_" name prefix (Jukka Lehtosalo, PR 17782)
- Support `Annotated[...]` in new-style type aliases (Jukka Lehtosalo, PR 17777)
- Fix nested generic classes (Jukka Lehtosalo, PR 17776)
- Add detection and error reporting for the use of incorrect expressions within the scope of a type parameter and a type alias (Kirill Podoprigora, PR 17560)

Basic Support for Python 3.13

This release adds partial support for Python 3.13 features and compiled binaries for Python 3.13. Mypyc now also supports Python 3.13.

In particular, these features are supported:

- Various new stdlib features and changes (through typeshed stub improvements)
- `typing.ReadOnly` (see below for more)
- `typing.TypeIs` (added in mypy 1.10, PEP 742)
- Type parameter defaults when using the legacy syntax (PEP 696)

These features are not supported yet:

- `warnings.deprecated` (PEP 702)
- Type parameter defaults when using Python 3.12 type parameter syntax

Mypyc Support for Python 3.13

Mypyc now supports Python 3.13. This was contributed by Marc Mueller, with additional fixes by Jukka Lehtosalo. Free threaded Python 3.13 builds are not supported yet.

List of changes:

- Add additional includes for Python 3.13 (Marc Mueller, PR 17506)
- Add another include for Python 3.13 (Marc Mueller, PR 17509)
- Fix ManagedDict functions for Python 3.13 (Marc Mueller, PR 17507)
- Update mypyc test output for Python 3.13 (Marc Mueller, PR 17508)
- Fix `PyUnicode` functions for Python 3.13 (Marc Mueller, PR 17504)
- Fix `_PyObject_LookupAttrId` for Python 3.13 (Marc Mueller, PR 17505)
- Fix `_PyList_Extend` for Python 3.13 (Marc Mueller, PR 17503)
- Fix `gen_is_coroutine` for Python 3.13 (Marc Mueller, PR 17501)
- Fix `_PyObject_FastCall` for Python 3.13 (Marc Mueller, PR 17502)
- Avoid uses of `_PyObject_CallMethodOneArg` on 3.13 (Jukka Lehtosalo, PR 17526)

- Don't rely on `_PyType_CalculateMetaclass` on 3.13 (Jukka Lehtosalo, PR 17525)
- Don't use `_PyUnicode_FastCopyCharacters` on 3.13 (Jukka Lehtosalo, PR 17524)
- Don't use `_PyUnicode_EQ` on 3.13, as it's no longer exported (Jukka Lehtosalo, PR 17523)

Inferring Unions for Conditional Expressions

Mypy now always tries to infer a union type for a conditional expression if left and right operand types are different. This results in more precise inferred types and lets mypy detect more issues. Example:

```
s = "foo" if cond() else 1
# Type of "s" is now "str | int" (it used to be "object")
```

Notably, if one of the operands has type `Any`, the type of a conditional expression is now `<type> | Any`. Previously the inferred type was just `Any`. The new type essentially indicates that the value can be of type `<type>`, and potentially of some (unknown) type. Most operations performed on the result must also be valid for `<type>`. Example where this is relevant:

```
from typing import Any

def func(a: Any, b: bool) -> None:
    x = a if b else None
    # Type of x is "Any | None"
    print(x.y)  # Error: None has no attribute "y"
```

This feature was contributed by Ivan Levkivskyi (PR 17427).

ReadOnly Support for TypedDict (PEP 705)

You can now use `typing.ReadOnly` to specity TypedDict items as read-only (PEP 705):

```
from typing import TypedDict

# Or "from typing ..." on Python 3.13
from typing_extensions import ReadOnly

class TD(TypedDict):
    a: int
    b: ReadOnly[int]

d: TD = {"a": 1, "b": 2}
d["a"] = 3  # OK
d["b"] = 5  # Error: "b" is ReadOnly
```

This feature was contributed by Nikita Sobolev (PR 17644).

Python 3.8 End of Life Approaching

We are planning to drop support for Python 3.8 in the next mypy feature release or the one after that. Python 3.8 reaches end of life in October 2024.

Planned Changes to Defaults

We are planning to enable `--local-partial-types` by default in mypy 2.0. This will often require at least minor code changes. This option is implicitly enabled by mypy daemon, so this makes the behavior of daemon and non-daemon modes consistent.

We recommend that mypy users start using local partial types soon (or to explicitly disable them) to prepare for the change.

This can also be configured in a mypy configuration file:

```
local_partial_types = True
```

For more information, refer to the documentation.

Documentation Updates

Mypy documentation now uses modern syntax variants and imports in many examples. Some examples no longer work on Python 3.8, which is the earliest Python version that mypy supports.

Notably, `Iterable` and other protocols/ABCs are imported from `collections.abc` instead of `typing`:

```
from collections.abc import Iterable, Callable
```

Examples also avoid the upper-case aliases to built-in types: `list[str]` is used instead of `List[str]`. The `X | Y` union type syntax introduced in Python 3.10 is also now prevalent.

List of documentation updates:

- Document `--output=json` CLI option (Edgar Ramírez Mondragón, PR 17611)
- Update various references to deprecated type aliases in docs (Jukka Lehtosalo, PR 17829)
- Make "X | Y" union syntax more prominent in documentation (Jukka Lehtosalo, PR 17835)
- Discuss upper bounds before self types in documentation (Jukka Lehtosalo, PR 17827)
- Make changelog visible in mypy documentation (quinn-sasha, PR 17742)
- List all incomplete features in `--enable-incomplete-feature` docs (sobolevn, PR 17633)
- Remove the explicit setting of a pygments theme (Pradyun Gedam, PR 17571)
- Document ReadOnly with TypedDict (Jukka Lehtosalo, PR 17905)
- Document TypeIs (Chelsea Durazo, PR 17821)

Experimental Inline TypedDict Syntax

Mypy now supports a non-standard, experimental syntax for defining anonymous TypedDicts. Example:

```
def func(n: str, y: int) -> {"name": str, "year": int}:
    return {"name": n, "year": y}
```

The feature is disabled by default. Use `--enable-incomplete-feature=InlineTypedDict` to enable it. *We might remove this feature in a future release.*

This feature was contributed by Ivan Levkivskyi (PR 17457).

Stubgen Improvements

- Fix crash on literal class-level keywords (sobolevn, PR 17663)
- Stubgen add `--version` (sobolevn, PR 17662)
- Fix `stubgen --no-analysis/--parse-only` docs (sobolevn, PR 17632)
- Include keyword only args when generating signatures in stubgenc (Eric Mark Martin, PR 17448)
- Add support for detecting `Literal` types when extracting types from docstrings (Michael Carlstrom, PR 17441)
- Use `Generator` type var defaults (Sebastian Rittau, PR 17670)

Stubtest Improvements

- Add support for `cached_property` (Ali Hamdan, PR 17626)
- Add `enable_incomplete_feature` validation to `stubtest` (sobolevn, PR 17635)
- Fix error code handling in `stubtest` with `--mypy-config-file` (sobolevn, PR 17629)

Other Notables Fixes and Improvements

- Report error if using unsupported type parameter defaults (Jukka Lehtosalo, PR 17876)
- Fix re-processing cross-reference in mypy daemon when node kind changes (Ivan Levkivskyi, PR 17883)
- Don't use equality to narrow when value is IntEnum/StrEnum (Jukka Lehtosalo, PR 17866)
- Don't consider None vs IntEnum comparison ambiguous (Jukka Lehtosalo, PR 17877)
- Fix narrowing of IntEnum and StrEnum types (Jukka Lehtosalo, PR 17874)
- Filter overload items based on self type during type inference (Jukka Lehtosalo, PR 17873)
- Enable negative narrowing of union TypeVar upper bounds (Brian Schubert, PR 17850)
- Fix issue with member expression formatting (Brian Schubert, PR 17848)
- Avoid type size explosion when expanding types (Jukka Lehtosalo, PR 17842)
- Fix negative narrowing of tuples in match statement (Brian Schubert, PR 17817)
- Narrow falsey str/bytes/int to literal type (Brian Schubert, PR 17818)
- Test against latest Python 3.13, make testing 3.14 easy (Shantanu, PR 17812)
- Reject ParamSpec-typed callables calls with insufficient arguments (Stanislav Terliakov, PR 17323)
- Fix crash when passing too many type arguments to generic base class accepting single ParamSpec (Brian Schubert, PR 17770)
- Fix TypeVar upper bounds sometimes not being displayed in pretty callables (Brian Schubert, PR 17802)
- Added error code for overlapping function signatures (Katrina Connors, PR 17597)
- Check for `truthy-bool` in `not ...` unary expressions (sobolevn, PR 17773)
- Add missing lines-covered and lines-valid attributes (Soubhik Kumar Mitra, PR 17738)
- Fix another crash scenario with recursive tuple types (Ivan Levkivskyi, PR 17708)
- Resolve TypeVar upper bounds in `functools.partial` (Shantanu, PR 17660)
- Always reset binder when checking deferred nodes (Ivan Levkivskyi, PR 17643)
- Fix crash on a callable attribute with single unpack (Ivan Levkivskyi, PR 17641)

- Fix mismatched signature between checker plugin API and implementation (bzoracler, PR 17343)
- Indexing a type also produces a GenericAlias (Shantanu, PR 17546)
- Fix crash on self-type in callable protocol (Ivan Levkivskyi, PR 17499)
- Fix crash on NamedTuple with method and error in function (Ivan Levkivskyi, PR 17498)
- Add `__replace__` for dataclasses in 3.13 (Max Muoto, PR 17469)
- Fix help message for `--no-namespace-packages` (Raphael Krupinski, PR 17472)
- Fix typechecking for async generators (Danny Yang, PR 17452)
- Fix strict optional handling in attrs plugin (Ivan Levkivskyi, PR 17451)
- Allow mixing ParamSpec and TypeVarTuple in Generic (Ivan Levkivskyi, PR 17450)
- Improvements to `functools.partial` of types (Shantanu, PR 17898)
- Make ReadOnly TypedDict items covariant (Jukka Lehtosalo, PR 17904)
- Fix union callees with `functools.partial` (Jukka Lehtosalo, PR 17903)
- Improve handling of generic functions with `functools.partial` (Ivan Levkivskyi, PR 17925)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Mypy 1.12.1

- Fix crash when showing partially analyzed type in error message (Ivan Levkivskyi, PR 17961)
- Fix iteration over union (when self type is involved) (Shantanu, PR 17976)
- Fix type object with type var default in union context (Jukka Lehtosalo, PR 17991)
- Revert change to `os.path` stubs affecting use of `os.PathLike[Any]` (Shantanu, PR 17995)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Ali Hamdan
- Anders Kaseorg
- Bénédikt Tran
- Brian Schubert
- bzoracler
- Chelsea Durazo
- Danny Yang
- Edgar Ramírez Mondragón
- Eric Mark Martin
- InSync
- Ivan Levkivskyi
- Jordandev678
- Katrina Connors

- Kirill Podoprigora
- Marc Mueller
- Max Muoto
- Max Murin
- Michael Carlstrom
- Michael I Chen
- Pradyun Gedam
- quinn-sasha
- Raphael Krupinski
- Sebastian Rittau
- Shantanu
- sobolevn
- Soubhik Kumar Mitra
- Stanislav Terliakov
- wyattscarpenter

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.13 Mypy 1.11

We've just uploaded mypy 1.11 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Support Python 3.12 Syntax for Generics (PEP 695)

Mypy now supports the new type parameter syntax introduced in Python 3.12 (PEP 695). This feature is still experimental and must be enabled with the `--enable-incomplete-feature=NewGenericSyntax` flag, or with `enable_incomplete_feature = NewGenericSyntax` in the mypy configuration file. We plan to enable this by default in the next mypy feature release.

This example demonstrates the new syntax:

```
# Generic function
def f[T](x: T) -> T: ...

reveal_type(f(1))  # Revealed type is 'int'

# Generic class
class C[T]:
    def __init__(self, x: T) -> None:
       self.x = x

c = C('a')
reveal_type(c.x)  # Revealed type is 'str'
```

(continues on next page)

(continued from previous page)

```
# Type alias
type A[T] = C[list[T]]
```

This feature was contributed by Jukka Lehtosalo.

Support for `functools.partial`

Mypy now type checks uses of `functools.partial`. Previously mypy would accept arbitrary arguments.

This example will now produce an error:

```
from functools import partial

def f(a: int, b: str) -> None: ...

g = partial(f, 1)

# Argument has incompatible type "int"; expected "str"
g(11)
```

This feature was contributed by Shantanu (PR 16939).

Stricter Checks for Untyped Overrides

Past mypy versions didn't check if untyped methods were compatible with overridden methods. This would result in false negatives. Now mypy performs these checks when using `--check-untyped-defs`.

For example, this now generates an error if using `--check-untyped-defs`:

```
class Base:
    def f(self, x: int = 0) -> None: ...

class Derived(Base):
    # Signature incompatible with "Base"
    def f(self): ...
```

This feature was contributed by Steven Troxler (PR 17276).

Type Inference Improvements

The new polymorphic inference algorithm introduced in mypy 1.5 is now used in more situations. This improves type inference involving generic higher-order functions, in particular.

This feature was contributed by Ivan Levkivskyi (PR 17348).

Mypy now uses unions of tuple item types in certain contexts to enable more precise inferred types. Example:

```
for x in (1, 'x'):
    # Previously inferred as 'object'
    reveal_type(x)  # Revealed type is 'int | str'
```

This was also contributed by Ivan Levkivskyi (PR 17408).

Improvements to Detection of Overlapping Overloads

The details of how mypy checks if two `@overload` signatures are unsafely overlapping were overhauled. This both fixes some false positives, and allows mypy to detect additional unsafe signatures.

This feature was contributed by Ivan Levkivskyi (PR 17392).

Better Support for Type Hints in Expressions

Mypy now allows more expressions that evaluate to valid type annotations in all expression contexts. The inferred types of these expressions are also sometimes more precise. Previously they were often `object`.

This example uses a union type that includes a callable type as an expression, and it no longer generates an error:

```
from typing import Callable

print(Callable[[], int] | None)  # No error
```

This feature was contributed by Jukka Lehtosalo (PR 17404).

Mypyc Improvements

Mypyc now supports the new syntax for generics introduced in Python 3.12 (see above). Another notable improvement is significantly faster basic operations on `int` values.

- Support Python 3.12 syntax for generic functions and classes (Jukka Lehtosalo, PR 17357)
- Support Python 3.12 type alias syntax (Jukka Lehtosalo, PR 17384)
- Fix ParamSpec (Shantanu, PR 17309)
- Inline fast paths of integer unboxing operations (Jukka Lehtosalo, PR 17266)
- Inline tagged integer arithmetic and bitwise operations (Jukka Lehtosalo, PR 17265)
- Allow specifying primitives as pure (Jukka Lehtosalo, PR 17263)

Changes to Stubtest

- Ignore `_ios_support` (Alex Waygood, PR 17270)
- Improve support for Python 3.13 (Shantanu, PR 17261)

Changes to Stubgen

- Gracefully handle invalid `Optional` and recognize aliases to PEP 604 unions (Ali Hamdan, PR 17386)
- Fix for Python 3.13 (Jelle Zijlstra, PR 17290)
- Preserve enum value initialisers (Shantanu, PR 17125)

Miscellaneous New Features

- Add error format support and JSON output option via `--output json` (Tushar Sadhwani, PR 11396)
- Support `enum.member` in Python 3.11+ (Nikita Sobolev, PR 17382)
- Support `enum.nonmember` in Python 3.11+ (Nikita Sobolev, PR 17376)
- Support `namedtuple.__replace__` in Python 3.13 (Shantanu, PR 17259)
- Support `rename=True` in collections.namedtuple (Jelle Zijlstra, PR 17247)
- Add support for `__spec__` (Shantanu, PR 14739)

Changes to Error Reporting

- Mention `--enable-incomplete-feature=NewGenericSyntax` in messages (Shantanu, PR 17462)
- Do not report plugin-generated methods with `explicit-override` (sobolevn, PR 17433)
- Use and display namespaces for function type variables (Ivan Levkivskyi, PR 17311)
- Fix false positive for Final local scope variable in Protocol (GiorgosPapoutsakis, PR 17308)
- Use Never in more messages, use ambiguous in join (Shantanu, PR 17304)
- Log full path to config file in verbose output (dexterkennedy, PR 17180)
- Added `[prop-decorator]` code for unsupported property decorators (#14461) (Christopher Barber, PR 16571)
- Suppress second error message with `:=` and `[truthy-bool]` (Nikita Sobolev, PR 15941)
- Generate error for assignment of functional Enum to variable of different name (Shantanu, PR 16805)
- Fix error reporting on cached run after uninstallation of third party library (Shantanu, PR 17420)

Fixes for Crashes

- Fix daemon crash on invalid type in TypedDict (Ivan Levkivskyi, PR 17495)
- Fix crash and bugs related to `partial()` (Ivan Levkivskyi, PR 17423)
- Fix crash when overriding with unpacked TypedDict (Ivan Levkivskyi, PR 17359)
- Fix crash on TypedDict unpacking for ParamSpec (Ivan Levkivskyi, PR 17358)
- Fix crash involving recursive union of tuples (Ivan Levkivskyi, PR 17353)
- Fix crash on invalid callable property override (Ivan Levkivskyi, PR 17352)
- Fix crash on unpacking self in NamedTuple (Ivan Levkivskyi, PR 17351)
- Fix crash on recursive alias with an optional type (Ivan Levkivskyi, PR 17350)
- Fix crash on type comment inside generic definitions (Bénédikt Tran, PR 16849)

Changes to Documentation

- Use inline config in documentation for optional error codes (Shantanu, PR 17374)
- Use lower-case generics in documentation (Seo Sanghyeon, PR 17176)
- Add documentation for show-error-code-links (GiorgosPapoutsakis, PR 17144)
- Update CONTRIBUTING.md to include commands for Windows (GiorgosPapoutsakis, PR 17142)

Other Notable Improvements and Fixes

- Fix ParamSpec inference against TypeVarTuple (Ivan Levkivskyi, PR 17431)
- Fix explicit type for `partial` (Ivan Levkivskyi, PR 17424)
- Always allow lambda calls (Ivan Levkivskyi, PR 17430)
- Fix isinstance checks with PEP 604 unions containing None (Shantanu, PR 17415)
- Fix self-referential upper bound in new-style type variables (Ivan Levkivskyi, PR 17407)
- Consider overlap between instances and callables (Ivan Levkivskyi, PR 17389)
- Allow new-style self-types in classmethods (Ivan Levkivskyi, PR 17381)
- Fix isinstance with type aliases to PEP 604 unions (Shantanu, PR 17371)

- Properly handle unpacks in overlap checks (Ivan Levkivskyi, PR 17356)
- Fix type application for classes with generic constructors (Ivan Levkivskyi, PR 17354)
- Update `typing_extensions` to >=4.6.0 to fix Python 3.12 error (Ben Brown, PR 17312)
- Avoid "does not return" error in lambda (Shantanu, PR 17294)
- Fix bug with descriptors in non-strict-optional mode (Max Murin, PR 17293)
- Don't leak unreachability from lambda body to surrounding scope (Anders Kaseorg, PR 17287)
- Fix issues with non-ASCII characters on Windows (Alexander Leopold Shon, PR 17275)
- Fix for type narrowing of negative integer literals (gilesgc, PR 17256)
- Fix confusion between .py and .pyi files in mypy daemon (Valentin Stanciu, PR 17245)
- Fix type of `tuple[X, Y]` expression (urnest, PR 17235)
- Don't forget that a `TypedDict` was wrapped in `Unpack` after a `name-defined` error occurred (Christoph Tyralla, PR 17226)
- Mark annotated argument as having an explicit, not inferred type (bzoracler, PR 17217)
- Don't consider Enum private attributes as enum members (Ali Hamdan, PR 17182)
- Fix Literal strings containing pipe characters (Jelle Zijlstra, PR 17148)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Mypy 1.11.1

- Fix `RawExpressionType.accept` crash with `--cache-fine-grained` (Anders Kaseorg, PR 17588)
- Fix PEP 604 isinstance caching (Shantanu, PR 17563)
- Fix `typing.TypeAliasType` being undefined on python < 3.12 (Nikita Sobolev, PR 17558)
- Fix `types.GenericAlias` lookup crash (Shantanu, PR 17543)

Mypy 1.11.2

- Alternative fix for a union-like literal string (Ivan Levkivskyi, PR 17639)
- Unwrap `TypedDict` item types before storing (Ivan Levkivskyi, PR 17640)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alex Waygood
- Alexander Leopold Shon
- Ali Hamdan
- Anders Kaseorg
- Ben Brown
- Bénédikt Tran
- bzoracler
- Christoph Tyralla

- Christopher Barber
- dexterkennedy
- gilesgc
- GiorgosPapoutsakis
- Ivan Levkivskyi
- Jelle Zijlstra
- Jukka Lehtosalo
- Marc Mueller
- Matthieu Devlin
- Michael R. Crusoe
- Nikita Sobolev
- Seo Sanghyeon
- Shantanu
- sobolevn
- Steven Troxler
- Tadeu Manoel
- Tamir Duberstein
- Tushar Sadhwani
- urnest
- Valentin Stanciu

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.14 Mypy 1.10

We've just uploaded mypy 1.10 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Support TypeIs (PEP 742)

Mypy now supports `TypeIs` (PEP 742), which allows functions to narrow the type of a value, similar to `isinstance()`. Unlike `TypeGuard`, `TypeIs` can narrow in both the `if` and `else` branches of an if statement:

```
from typing_extensions import TypeIs

def is_str(s: object) -> TypeIs[str]:
    return isinstance(s, str)

def f(o: str | int) -> None:
    if is_str(o):
        # Type of o is 'str'
```

(continues on next page)

(continued from previous page)

```
        ...
    else:
        # Type of o is 'int'
        ...
```

TypeIs will be added to the typing module in Python 3.13, but it can be used on earlier Python versions by importing it from typing_extensions.

This feature was contributed by Jelle Zijlstra (PR 16898).

Support TypeVar Defaults (PEP 696)

PEP 696 adds support for type parameter defaults. Example:

```
from typing import Generic
from typing_extensions import TypeVar

T = TypeVar("T", default=int)

class C(Generic[T]):
   ...

x: C = ...
y: C[str] = ...
reveal_type(x)  # C[int], because of the default
reveal_type(y)  # C[str]
```

TypeVar defaults will be added to the typing module in Python 3.13, but they can be used with earlier Python releases by importing TypeVar from typing_extensions.

This feature was contributed by Marc Mueller (PR 16878 and PR 16925).

Support TypeAliasType (PEP 695)

As part of the initial steps towards implementing PEP 695, mypy now supports TypeAliasType. TypeAliasType provides a backport of the new type statement in Python 3.12.

```
type ListOrSet[T] = list[T] | set[T]
```

is equivalent to:

```
T = TypeVar("T")
ListOrSet = TypeAliasType("ListOrSet", list[T] | set[T], type_params=(T,))
```

Example of use in mypy:

```
from typing_extensions import TypeAliasType, TypeVar

NewUnionType = TypeAliasType("NewUnionType", int | str)
x: NewUnionType = 42
y: NewUnionType = 'a'
z: NewUnionType = object()  # error: Incompatible types in assignment (expression has␣
↪type "object", variable has type "int | str")  [assignment]
```

(continues on next page)

(continued from previous page)

```
T = TypeVar("T")
ListOrSet = TypeAliasType("ListOrSet", list[T] | set[T], type_params=(T,))
a: ListOrSet[int] = [1, 2]
b: ListOrSet[str] = {'a', 'b'}
c: ListOrSet[str] = 'test'  # error: Incompatible types in assignment (expression has␣
↪type "str", variable has type "list[str] | set[str]")  [assignment]
```

TypeAliasType was added to the typing module in Python 3.12, but it can be used with earlier Python releases by importing from typing_extensions.

This feature was contributed by Ali Hamdan (PR 16926, PR 17038 and PR 17053)

Detect Additional Unsafe Uses of super()

Mypy will reject unsafe uses of super() more consistently, when the target has a trivial (empty) body. Example:

```
class Proto(Protocol):
    def method(self) -> int: ...

class Sub(Proto):
    def method(self) -> int:
        return super().meth()  # Error (unsafe)
```

This feature was contributed by Shantanu (PR 16756).

Stubgen Improvements

- Preserve empty tuple annotation (Ali Hamdan, PR 16907)
- Add support for PEP 570 positional-only parameters (Ali Hamdan, PR 16904)
- Replace obsolete typing aliases with builtin containers (Ali Hamdan, PR 16780)
- Fix generated dataclass __init__ signature (Ali Hamdan, PR 16906)

Mypyc Improvements

- Provide an easier way to define IR-to-IR transforms (Jukka Lehtosalo, PR 16998)
- Implement lowering pass and add primitives for int (in)equality (Jukka Lehtosalo, PR 17027)
- Implement lowering for remaining tagged integer comparisons (Jukka Lehtosalo, PR 17040)
- Optimize away some bool/bit registers (Jukka Lehtosalo, PR 17022)
- Remangle redefined names produced by async with (Richard Si, PR 16408)
- Optimize TYPE_CHECKING to False at Runtime (Srinivas Lade, PR 16263)
- Fix compilation of unreachable comprehensions (Richard Si, PR 15721)
- Don't crash on non-inlinable final local reads (Richard Si, PR 15719)

Documentation Improvements

- Import TypedDict from typing instead of typing_extensions (Riccardo Di Maio, PR 16958)
- Add missing mutable-override to section title (James Braza, PR 16886)

Error Reporting Improvements

- Use lower-case generics more consistently in error messages (Jukka Lehtosalo, PR 17035)

Other Notable Changes and Fixes

- Fix incorrect inferred type when accessing descriptor on union type (Matthieu Devlin, PR 16604)
- Fix crash when expanding invalid `Unpack` in a `Callable` alias (Ali Hamdan, PR 17028)
- Fix false positive when string formatting with string enum (roberfi, PR 16555)
- Narrow individual items when matching a tuple to a sequence pattern (Loïc Simon, PR 16905)
- Fix false positive from type variable within TypeGuard or TypeIs (Evgeniy Slobodkin, PR 17071)
- Improve `yield from` inference for unions of generators (Shantanu, PR 16717)
- Fix emulating hash method logic in `attrs` classes (Hashem, PR 17016)
- Add reverted typeshed commit that uses `ParamSpec` for `functools.wraps` (Tamir Duberstein, PR 16942)
- Fix type narrowing for `types.EllipsisType` (Shantanu, PR 17003)
- Fix single item enum match type exhaustion (Oskari Lehto, PR 16966)
- Improve type inference with empty collections (Marc Mueller, PR 16994)
- Fix override checking for decorated property (Shantanu, PR 16856)
- Fix narrowing on match with function subject (Edward Paget, PR 16503)
- Allow `+N` within `Literal[...]` (Spencer Brown, PR 16910)
- Experimental: Support TypedDict within `type[...]` (Marc Mueller, PR 16963)
- Experimtental: Fix issue with TypedDict with optional keys in `type[...]` (Marc Mueller, PR 17068)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Mypy 1.10.1

- Fix error reporting on cached run after uninstallation of third party library (Shantanu, PR 17420)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alex Waygood
- Ali Hamdan
- Edward Paget
- Evgeniy Slobodkin
- Hashem
- hesam
- Hugo van Kemenade
- Ihor
- James Braza

- Jelle Zijlstra
- jhance
- Jukka Lehtosalo
- Loïc Simon
- Marc Mueller
- Matthieu Devlin
- Michael R. Crusoe
- Nikita Sobolev
- Oskari Lehto
- Riccardo Di Maio
- Richard Si
- roberfi
- Roman Solomatin
- Sam Xifaras
- Shantanu
- Spencer Brown
- Srinivas Lade
- Tamir Duberstein
- youkaichao

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.15 Mypy 1.9

We've just uploaded mypy 1.9 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Breaking Changes

Because the version of typeshed we use in mypy 1.9 doesn't support 3.7, neither does mypy 1.9. (Jared Hance, PR 16883)

We are planning to enable local partial types (enabled via the `--local-partial-types` flag) later this year by default. This change was announced years ago, but now it's finally happening. This is a major backward-incompatible change, so we'll probably include it as part of the upcoming mypy 2.0 release. This makes daemon and non-daemon mypy runs have the same behavior by default.

Local partial types can also be enabled in the mypy config file:

```
local_partial_types = True
```

We are looking at providing a tool to make it easier to migrate projects to use `--local-partial-types`, but it's not yet clear whether this is practical. The migration usually involves adding some explicit type annotations to module-level and class-level variables.

Basic Support for Type Parameter Defaults (PEP 696)

This release contains new experimental support for type parameter defaults (PEP 696). Please try it out! This feature was contributed by Marc Mueller.

Since this feature will be officially introduced in the next Python feature release (3.13), you will need to import `TypeVar`, `ParamSpec` or `TypeVarTuple` from `typing_extensions` to use defaults for now.

This example adapted from the PEP defines a default for `BotT`:

```
from typing import Generic
from typing_extensions import TypeVar

class Bot: ...

BotT = TypeVar("BotT", bound=Bot, default=Bot)

class Context(Generic[BotT]):
    bot: BotT

class MyBot(Bot): ...

# type is Bot (the default)
reveal_type(Context().bot)
# type is MyBot
reveal_type(Context[MyBot]().bot)
```

Type-checking Improvements

- Fix missing type store for overloads (Marc Mueller, PR 16803)
- Fix `'WriteToConn' object has no attribute 'flush'` (Charlie Denton, PR 16801)
- Improve TypeAlias error messages (Marc Mueller, PR 16831)
- Support narrowing unions that include `type[None]` (Christoph Tyralla, PR 16315)
- Support TypedDict functional syntax as class base type (anniel-stripe, PR 16703)
- Accept multiline quoted annotations (Shantanu, PR 16765)
- Allow unary + in `Literal` (Jelle Zijlstra, PR 16729)
- Substitute type variables in return type of static methods (Kouroche Bouchiat, PR 16670)
- Consider TypeVarTuple to be invariant (Marc Mueller, PR 16759)
- Add `alias` support to `field()` in `attrs` plugin (Nikita Sobolev, PR 16610)
- Improve attrs hashability detection (Tin Tvrtković, PR 16556)

Performance Improvements

- Speed up finding function type variables (Jukka Lehtosalo, PR 16562)

Documentation Updates

- Document supported values for `--enable-incomplete-feature` in "mypy –help" (Froger David, PR 16661)
- Update new type system discussion links (thomaswhaley, PR 16841)
- Add missing class instantiation to cheat sheet (Aleksi Tarvainen, PR 16817)
- Document how evil `--no-strict-optional` is (Shantanu, PR 16731)
- Improve mypy daemon documentation note about local partial types (Makonnen Makonnen, PR 16782)
- Fix numbering error (Stefanie Molin, PR 16838)
- Various documentation improvements (Shantanu, PR 16836)

Stubtest Improvements

- Ignore private function/method parameters when they are missing from the stub (private parameter names start with a single underscore and have a default) (PR 16507)
- Ignore a new protocol dunder (Alex Waygood, PR 16895)
- Private parameters can be omitted (Sebastian Rittau, PR 16507)
- Add support for setting enum members to "..." (Jelle Zijlstra, PR 16807)
- Adjust symbol table logic (Shantanu, PR 16823)
- Fix posisitional-only handling in overload resolution (Shantanu, PR 16750)

Stubgen Improvements

- Fix crash on star unpack of TypeVarTuple (Ali Hamdan, PR 16869)
- Use PEP 604 unions everywhere (Ali Hamdan, PR 16519)
- Do not ignore property deleter (Ali Hamdan, PR 16781)
- Support type stub generation for `staticmethod` (WeilerMarcel, PR 14934)

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Aleksi Tarvainen
- Alex Waygood
- Ali Hamdan
- anniel-stripe
- Charlie Denton
- Christoph Tyralla
- Dheeraj
- Fabian Keller
- Fabian Lewis
- Froger David
- Ihor
- Jared Hance

- Jelle Zijlstra
- Jukka Lehtosalo
- Kouroche Bouchiat
- Lukas Geiger
- Maarten Huijsmans
- Makonnen Makonnen
- Marc Mueller
- Nikita Sobolev
- Sebastian Rittau
- Shantanu
- Stefanie Molin
- Stephen Morton
- thomaswhaley
- Tin Tvrtković
- WeilerMarcel
- Wesley Collin Wright
- zipperer

I'd also like to thank my employer, Dropbox, for supporting mypy development.

1.36.16 Mypy 1.8

We've just uploaded mypy 1.8 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Type-checking Improvements

- Do not intersect types in isinstance checks if at least one is final (Christoph Tyralla, PR 16330)
- Detect that `@final` class without `__bool__` cannot have falsey instances (Ilya Priven, PR 16566)
- Do not allow `TypedDict` classes with extra keywords (Nikita Sobolev, PR 16438)
- Do not allow class-level keywords for `NamedTuple` (Nikita Sobolev, PR 16526)
- Make imprecise constraints handling more robust (Ivan Levkivskyi, PR 16502)
- Fix strict-optional in extending generic TypedDict (Ivan Levkivskyi, PR 16398)
- Allow type ignores of PEP 695 constructs (Shantanu, PR 16608)
- Enable `type_check_only` support for `TypedDict` and `NamedTuple` (Nikita Sobolev, PR 16469)

Performance Improvements

- Add fast path to analyzing special form assignments (Jukka Lehtosalo, PR 16561)

Improvements to Error Reporting

- Don't show documentation links for plugin error codes (Ivan Levkivskyi, PR 16383)
- Improve error messages for `super` checks and add more tests (Nikita Sobolev, PR 16393)
- Add error code for mutable covariant override (Ivan Levkivskyi, PR 16399)

Stubgen Improvements

- Preserve simple defaults in function signatures (Ali Hamdan, PR 15355)
- Include `__all__` in output (Jelle Zijlstra, PR 16356)
- Fix stubgen regressions with pybind11 and mypy 1.7 (Chad Dombrova, PR 16504)

Stubtest Improvements

- Improve handling of unrepresentable defaults (Jelle Zijlstra, PR 16433)
- Print more helpful errors if a function is missing from stub (Alex Waygood, PR 16517)
- Support `@type_check_only` decorator (Nikita Sobolev, PR 16422)
- Warn about missing `__del__` (Shantanu, PR 16456)
- Fix crashes with some uses of `final` and `deprecated` (Shantanu, PR 16457)

Fixes to Crashes

- Fix crash with type alias to `Callable[[Unpack[Tuple[Any, ...]]], Any]` (Alex Waygood, PR 16541)
- Fix crash on TypeGuard in `__call__` (Ivan Levkivskyi, PR 16516)
- Fix crash on invalid enum in method (Ivan Levkivskyi, PR 16511)
- Fix crash on unimported Any in TypedDict (Ivan Levkivskyi, PR 16510)

Documentation Updates

- Update soft-error-limit default value to -1 (Sveinung Gundersen, PR 16542)
- Support Sphinx 7.x (Michael R. Crusoe, PR 16460)

Other Notable Changes and Fixes

- Allow mypy to output a junit file with per-file results (Matthew Wright, PR 16388)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alex Waygood
- Ali Hamdan

- Chad Dombrova
- Christoph Tyralla
- Ilya Priven
- Ivan Levkivskyi
- Jelle Zijlstra
- Jukka Lehtosalo
- Marcel Telka
- Matthew Wright
- Michael R. Crusoe
- Nikita Sobolev
- Ole Peder Brandtzæg
- robjhornby
- Shantanu
- Sveinung Gundersen
- Valentin Stanciu

I'd also like to thank my employer, Dropbox, for supporting mypy development.

Posted by Wesley Collin Wright

1.36.17 Mypy 1.7

We've just uploaded mypy 1.7 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Using TypedDict for **kwargs Typing

Mypy now has support for using `Unpack[...]` with a TypedDict type to annotate `**kwargs` arguments enabled by default. Example:

```
# Or 'from typing_extensions import ...'
from typing import TypedDict, Unpack

class Person(TypedDict):
    name: str
    age: int

def foo(**kwargs: Unpack[Person]) -> None:
    ...

foo(name="x", age=1)  # Ok
foo(name=1)  # Error
```

The definition of `foo` above is equivalent to the one below, with keyword-only arguments `name` and `age`:

```
def foo(*, name: str, age: int) -> None:
    ...
```

Refer to PEP 692 for more information. Note that unlike in the current version of the PEP, mypy always treats signatures with Unpack[SomeTypedDict] as equivalent to their expanded forms with explicit keyword arguments, and there aren't special type checking rules for TypedDict arguments.

This was contributed by Ivan Levkivskyi back in 2022 (PR 13471).

TypeVarTuple Support Enabled (Experimental)

Mypy now has support for variadic generics (TypeVarTuple) enabled by default, as an experimental feature. Refer to PEP 646 for the details.

TypeVarTuple was implemented by Jared Hance and Ivan Levkivskyi over several mypy releases, with help from Jukka Lehtosalo.

Changes included in this release:

- Fix handling of tuple type context with unpacks (Ivan Levkivskyi, PR 16444)
- Handle TypeVarTuples when checking overload constraints (robjhornby, PR 16428)
- Enable Unpack/TypeVarTuple support (Ivan Levkivskyi, PR 16354)
- Fix crash on unpack call special-casing (Ivan Levkivskyi, PR 16381)
- Some final touches for variadic types support (Ivan Levkivskyi, PR 16334)
- Support PEP-646 and PEP-692 in the same callable (Ivan Levkivskyi, PR 16294)
- Support new * syntax for variadic types (Ivan Levkivskyi, PR 16242)
- Correctly handle variadic instances with empty arguments (Ivan Levkivskyi, PR 16238)
- Correctly handle runtime type applications of variadic types (Ivan Levkivskyi, PR 16240)
- Support variadic tuple packing/unpacking (Ivan Levkivskyi, PR 16205)
- Better support for variadic calls and indexing (Ivan Levkivskyi, PR 16131)
- Subtyping and inference of user-defined variadic types (Ivan Levkivskyi, PR 16076)
- Complete type analysis of variadic types (Ivan Levkivskyi, PR 15991)

New Way of Installing Mypyc Dependencies

If you want to install package dependencies needed by mypyc (not just mypy), you should now install mypy[mypyc] instead of just mypy:

```
python3 -m pip install -U 'mypy[mypyc]'
```

Mypy has many more users than mypyc, so always installing mypyc dependencies would often bring unnecessary dependencies.

This change was contributed by Shantanu (PR 16229).

New Rules for Re-exports

Mypy no longer considers an import such as import a.b as b as an explicit re-export. The old behavior was arguably inconsistent and surprising. This may impact some stub packages, such as older versions of types-six. You can change the import to from a import b as b, if treating the import as a re-export was intentional.

This change was contributed by Anders Kaseorg (PR 14086).

Improved Type Inference

The new type inference algorithm that was recently introduced to mypy (but was not enabled by default) is now enabled by default. It improves type inference of calls to generic callables where an argument is also a generic callable, in particular. You can use `--old-type-inference` to disable the new behavior.

The new algorithm can (rarely) produce different error messages, different error codes, or errors reported on different lines. This is more likely in cases where generic types were used incorrectly.

The new type inference algorithm was contributed by Ivan Levkivskyi. PR 16345 enabled it by default.

Narrowing Tuple Types Using len()

Mypy now can narrow tuple types using `len()` checks. Example:

```
def f(t: tuple[int, int] | tuple[int, int, int]) -> None:
    if len(t) == 2:
        a, b = t   # Ok
    ...
```

This feature was contributed by Ivan Levkivskyi (PR 16237).

More Precise Tuple Lengths (Experimental)

Mypy supports experimental, more precise checking of tuple type lengths through `--enable-incomplete-feature=PreciseTupleTypes`. Refer to the documentation for more information.

More generally, we are planning to use `--enable-incomplete-feature` to introduce experimental features that would benefit from community feedback.

This feature was contributed by Ivan Levkivskyi (PR 16237).

Mypy Changelog

We now maintain a changelog in the mypy Git repository. It mirrors the contents of mypy release blog posts. We will continue to also publish release blog posts. In the future, release blog posts will be created based on the changelog near a release date.

This was contributed by Shantanu (PR 16280).

Mypy Daemon Improvements

- Fix daemon crash caused by deleted submodule (Jukka Lehtosalo, PR 16370)
- Fix file reloading in dmypy with –export-types (Ivan Levkivskyi, PR 16359)
- Fix dmypy inspect on Windows (Ivan Levkivskyi, PR 16355)
- Fix dmypy inspect for namespace packages (Ivan Levkivskyi, PR 16357)
- Fix return type change to optional in generic function (Jukka Lehtosalo, PR 16342)
- Fix daemon false positives related to module-level `__getattr__` (Jukka Lehtosalo, PR 16292)
- Fix daemon crash related to ABCs (Jukka Lehtosalo, PR 16275)
- Stream dmypy output instead of dumping everything at the end (Valentin Stanciu, PR 16252)
- Make sure all dmypy errors are shown (Valentin Stanciu, PR 16250)

Mypyc Improvements

- Generate error on duplicate function definitions (Jukka Lehtosalo, PR 16309)
- Don't crash on unreachable statements (Jukka Lehtosalo, PR 16311)
- Avoid cyclic reference in nested functions (Jukka Lehtosalo, PR 16268)
- Fix direct `__dict__` access on inner functions in new Python (Shantanu, PR 16084)
- Make tuple packing and unpacking more efficient (Jukka Lehtosalo, PR 16022)

Improvements to Error Reporting

- Update starred expression error message to match CPython (Cibin Mathew, PR 16304)
- Fix error code of "Maybe you forgot to use await" note (Jelle Zijlstra, PR 16203)
- Use error code `[unsafe-overload]` for unsafe overloads, instead of `[misc]` (Randolf Scholz, PR 16061)
- Reword the error message related to void functions (Albert Tugushev, PR 15876)
- Represent bottom type as Never in messages (Shantanu, PR 15996)
- Add hint for AsyncIterator incompatible return type (Ilya Priven, PR 15883)
- Don't suggest stubs packages where the runtime package now ships with types (Alex Waygood, PR 16226)

Performance Improvements

- Speed up type argument checking (Jukka Lehtosalo, PR 16353)
- Add fast path for checking self types (Jukka Lehtosalo, PR 16352)
- Cache information about whether file is typeshed file (Jukka Lehtosalo, PR 16351)
- Skip expensive `repr()` in logging call when not needed (Jukka Lehtosalo, PR 16350)

Attrs and Dataclass Improvements

- `dataclass.replace`: Allow transformed classes (Ilya Priven, PR 15915)
- `dataclass.replace`: Fall through to typeshed signature (Ilya Priven, PR 15962)
- Document `dataclass_transform` behavior (Ilya Priven, PR 16017)
- `attrs`: Remove fields type check (Ilya Priven, PR 15983)
- `attrs`, `dataclasses`: Don't enforce slots when base class doesn't (Ilya Priven, PR 15976)
- Fix crash on dataclass field / property collision (Nikita Sobolev, PR 16147)

Stubgen Improvements

- Write stubs with utf-8 encoding (Jørgen Lind, PR 16329)
- Fix missing property setter in semantic analysis mode (Ali Hamdan, PR 16303)
- Unify C extension and pure python stub generators with object oriented design (Chad Dombrova, PR 15770)
- Multiple fixes to the generated imports (Ali Hamdan, PR 15624)
- Generate valid dataclass stubs (Ali Hamdan, PR 15625)

Fixes to Crashes

- Fix incremental mode crash on TypedDict in method (Ivan Levkivskyi, PR 16364)
- Fix crash on star unpack in TypedDict (Ivan Levkivskyi, PR 16116)
- Fix crash on malformed TypedDict in incremental mode (Ivan Levkivskyi, PR 16115)
- Fix crash with report generation on namespace packages (Shantanu, PR 16019)
- Fix crash when parsing error code config with typo (Shantanu, PR 16005)
- Fix `__post_init__()` internal error (Ilya Priven, PR 16080)

Documentation Updates

- Make it easier to copy commands from README (Hamir Mahal, PR 16133)
- Document and rename `[overload-overlap]` error code (Shantanu, PR 16074)
- Document `--force-uppercase-builtins` and `--force-union-syntax` (Nikita Sobolev, PR 16049)
- Document `force_union_syntax` and `force_uppercase_builtins` (Nikita Sobolev, PR 16048)
- Document we're not tracking relationships between symbols (Ilya Priven, PR 16018)

Other Notable Changes and Fixes

- Propagate narrowed types to lambda expressions (Ivan Levkivskyi, PR 16407)
- Avoid importing from `setuptools._distutils` (Shantanu, PR 16348)
- Delete recursive aliases flags (Ivan Levkivskyi, PR 16346)
- Properly use proper subtyping for callables (Ivan Levkivskyi, PR 16343)
- Use upper bound as inference fallback more consistently (Ivan Levkivskyi, PR 16344)
- Add `[unimported-reveal]` error code (Nikita Sobolev, PR 16271)
- Add `|=` and `|` operators support for `TypedDict` (Nikita Sobolev, PR 16249)
- Clarify variance convention for Parameters (Ivan Levkivskyi, PR 16302)
- Correctly recognize `typing_extensions.NewType` (Ganden Schaffner, PR 16298)
- Fix partially defined in the case of missing type maps (Shantanu, PR 15995)
- Use SPDX license identifier (Nikita Sobolev, PR 16230)
- Make `__qualname__` and `__module__` available in class bodies (Anthony Sottile, PR 16215)
- stubtest: Hint when args in stub need to be keyword-only (Alex Waygood, PR 16210)
- Tuple slice should not propagate fallback (Thomas Grainger, PR 16154)
- Fix cases of type object handling for overloads (Shantanu, PR 16168)
- Fix walrus interaction with empty collections (Ivan Levkivskyi, PR 16197)
- Use type variable bound when it appears as actual during inference (Ivan Levkivskyi, PR 16178)
- Use upper bounds as fallback solutions for inference (Ivan Levkivskyi, PR 16184)
- Special-case type inference of empty collections (Ivan Levkivskyi, PR 16122)
- Allow TypedDict unpacking in Callable types (Ivan Levkivskyi, PR 16083)
- Fix inference for overloaded `__call__` with generic self (Shantanu, PR 16053)

- Call dynamic class hook on generic classes (Petter Friberg, PR 16052)
- Preserve implicitly exported types via attribute access (Shantanu, PR 16129)
- Fix a stubtest bug (Alex Waygood)
- Fix `tuple[Any, ...]` subtyping (Shantanu, PR 16108)
- Lenient handling of trivial Callable suffixes (Ivan Levkivskyi, PR 15913)
- Add `add_overloaded_method_to_class` helper for plugins (Nikita Sobolev, PR 16038)
- Bundle `misc/proper_plugin.py` as a part of `mypy` (Nikita Sobolev, PR 16036)
- Fix `case Any()` in match statement (DS/Charlie, PR 14479)
- Make iterable logic more consistent (Shantanu, PR 16006)
- Fix inference for properties with `__call__` (Shantanu, PR 15926)

Typeshed Updates

Please see git log for full list of standard library typeshed stub changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Albert Tugushev
- Alex Waygood
- Ali Hamdan
- Anders Kaseorg
- Anthony Sottile
- Chad Dombrova
- Cibin Mathew
- dinaldoap
- DS/Charlie
- Eli Schwartz
- Ganden Schaffner
- Hamir Mahal
- Ihor
- Ikko Eltociear Ashimine
- Ilya Priven
- Ivan Levkivskyi
- Jelle Zijlstra
- Jukka Lehtosalo
- Jørgen Lind
- KotlinIsland
- Matt Bogosian

- Nikita Sobolev
- Petter Friberg
- Randolf Scholz
- Shantanu
- Thomas Grainger
- Valentin Stanciu

I'd also like to thank my employer, Dropbox, for supporting mypy development.

Posted by Jukka Lehtosalo

1.36.18 Mypy 1.6

Tuesday, 10 October 2023

We've just uploaded mypy 1.6 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Introduce Error Subcodes for Import Errors

Mypy now uses the error code import-untyped if an import targets an installed library that doesn't support static type checking, and no stub files are available. Other invalid imports produce the import-not-found error code. They both are subcodes of the import error code, which was previously used for both kinds of import-related errors.

Use –disable-error-code=import-untyped to only ignore import errors about installed libraries without stubs. This way mypy will still report errors about typos in import statements, for example.

If you use –warn-unused-ignore or –strict, mypy will complain if you use # type: ignore[import] to ignore an import error. You are expected to use one of the more specific error codes instead. Otherwise, ignoring the import error code continues to silence both errors.

This feature was contributed by Shantanu (PR 15840, PR 14740).

Remove Support for Targeting Python 3.6 and Earlier

Running mypy with –python-version 3.6, for example, is no longer supported. Python 3.6 hasn't been properly supported by mypy for some time now, and this makes it explicit. This was contributed by Nikita Sobolev (PR 15668).

Selective Filtering of –disallow-untyped-calls Targets

Using –disallow-untyped-calls could be annoying when using libraries with missing type information, as mypy would generate many errors about code that uses the library. Now you can use –untyped-calls-exclude=acme, for example, to disable these errors about calls targeting functions defined in the acme package. Refer to the documentation for more information.

This feature was contributed by Ivan Levkivskyi (PR 15845).

Improved Type Inference between Callable Types

Mypy now does a better job inferring type variables inside arguments of callable types. For example, this code fragment now type checks correctly:

```python
def f(c: Callable[[T, S], None]) -> Callable[[str, T, S], None]: ...
def g(*x: int) -> None: ...

reveal_type(f(g))  # Callable[[str, int, int], None]
```

This was contributed by Ivan Levkivskyi (PR 15910).

Don't Consider None and TypeVar to Overlap in Overloads

Mypy now doesn't consider an overload item with an argument type None to overlap with a type variable:

```python
@overload
def f(x: None) -> None: ..
@overload
def f(x: T) -> Foo[T]: ...
...
```

Previously mypy would generate an error about the definition of f above. This is slightly unsafe if the upper bound of T is object, since the value of the type variable could be None. We relaxed the rules a little, since this solves a common issue.

This feature was contributed by Ivan Levkivskyi (PR 15846).

Improvements to –new-type-inference

The experimental new type inference algorithm (polymorphic inference) introduced as an opt-in feature in mypy 1.5 has several improvements:

- Improve transitive closure computation during constraint solving (Ivan Levkivskyi, PR 15754)
- Add support for upper bounds and values with –new-type-inference (Ivan Levkivskyi, PR 15813)
- Basic support for variadic types with –new-type-inference (Ivan Levkivskyi, PR 15879)
- Polymorphic inference: support for parameter specifications and lambdas (Ivan Levkivskyi, PR 15837)
- Invalidate cache when adding –new-type-inference (Marc Mueller, PR 16059)

Note: We are planning to enable –new-type-inference by default in mypy 1.7. Please try this out and let us know if you encounter any issues.

ParamSpec Improvements

- Support self-types containing ParamSpec (Ivan Levkivskyi, PR 15903)
- Allow "…" in Concatenate, and clean up ParamSpec literals (Ivan Levkivskyi, PR 15905)
- Fix ParamSpec inference for callback protocols (Ivan Levkivskyi, PR 15986)
- Infer ParamSpec constraint from arguments (Ivan Levkivskyi, PR 15896)
- Fix crash on invalid type variable with ParamSpec (Ivan Levkivskyi, PR 15953)
- Fix subtyping between ParamSpecs (Ivan Levkivskyi, PR 15892)

Stubgen Improvements

- Add option to include docstrings with stubgen (chylek, PR 13284)
- Add required … initializer to NamedTuple fields with default values (Nikita Sobolev, PR 15680)

Stubtest Improvements

- Fix __mypy-replace false positives (Alex Waygood, PR 15689)
- Fix edge case for bytes enum subclasses (Alex Waygood, PR 15943)
- Generate error if typeshed is missing modules from the stdlib (Alex Waygood, PR 15729)
- Fixes to new check for missing stdlib modules (Alex Waygood, PR 15960)
- Fix stubtest enum.Flag edge case (Alex Waygood, PR 15933)

Documentation Improvements

- Do not advertise to create your own assert_never helper (Nikita Sobolev, PR 15947)
- Fix all the missing references found within the docs (Albert Tugushev, PR 15875)
- Document await-not-async error code (Shantanu, PR 15858)
- Improve documentation of disabling error codes (Shantanu, PR 15841)

Other Notable Changes and Fixes

- Make unsupported PEP 695 features (introduced in Python 3.12) give a reasonable error message (Shantanu, PR 16013)
- Remove the –py2 command-line argument (Marc Mueller, PR 15670)
- Change empty tuple from tuple[] to tuple[()] in error messages (Nikita Sobolev, PR 15783)
- Fix assert_type failures when some nodes are deferred (Nikita Sobolev, PR 15920)
- Generate error on unbound TypeVar with values (Nikita Sobolev, PR 15732)
- Fix over-eager types-google-cloud-ndb suggestion (Shantanu, PR 15347)
- Fix type narrowing of == None and in (None,) conditions (Marti Raudsepp, PR 15760)
- Fix inference for attrs.fields (Shantanu, PR 15688)
- Make "await in non-async function" a non-blocking error and give it an error code (Gregory Santosa, PR 15384)
- Add basic support for decorated overloads (Ivan Levkivskyi, PR 15898)
- Fix TypeVar regression with self types (Ivan Levkivskyi, PR 15945)
- Add __match_args__ to dataclasses with no fields (Ali Hamdan, PR 15749)
- Include stdout and stderr in dmypy verbose output (Valentin Stanciu, PR 15881)
- Improve match narrowing and reachability analysis (Shantanu, PR 15882)
- Support __bool__ with Literal in –warn-unreachable (Jannic Warken, PR 15645)
- Fix inheriting from generic @frozen attrs class (Ilya Priven, PR 15700)
- Correctly narrow types for tuple[type[X], ...] (Nikita Sobolev, PR 15691)
- Don't flag intentionally empty generators unreachable (Ilya Priven, PR 15722)
- Add tox.ini to mypy sdist (Marcel Telka, PR 15853)
- Fix mypyc regression with pretty (Shantanu, PR 16124)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to Max Murin, who did most of the release manager work for this release (I just did the final steps).

Thanks to all mypy contributors who contributed to this release:

- Albert Tugushev
- Alex Waygood
- Ali Hamdan
- chylek
- EXPLOSION
- Gregory Santosa
- Ilya Priven
- Ivan Levkivskyi
- Jannic Warken
- KotlinIsland
- Marc Mueller
- Marcel Johannesmann
- Marcel Telka
- Mark Byrne
- Marti Raudsepp
- Max Murin
- Nikita Sobolev
- Shantanu
- Valentin Stanciu

Posted by Jukka Lehtosalo

1.36.19 Mypy 1.5

Thursday, 10 August 2023

We've just uploaded mypy 1.5 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, deprecations and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Drop Support for Python 3.7

Mypy no longer supports running with Python 3.7, which has reached end-of-life. This was contributed by Shantanu (PR 15566).

Optional Check to Require Explicit @override

If you enable the explicit-override error code, mypy will generate an error if a method override doesn't use the @typing.override decorator (as discussed in PEP 698). This way mypy will detect accidentally introduced overrides. Example:

```
# mypy: enable-error-code="explicit-override"

from typing_extensions import override

class C:
    def foo(self) -> None: pass
    def bar(self) -> None: pass

class D(C):
    # Error: Method "foo" is not using @override but is
    # overriding a method
    def foo(self) -> None:
        ...

    @override
    def bar(self) -> None:  # OK
        ...
```

You can enable the error code via –enable-error-code=explicit-override on the mypy command line or enable_error_code = explicit-override in the mypy config file.

The override decorator will be available in typing in Python 3.12, but you can also use the backport from a recent version of `typing_extensions` on all supported Python versions.

This feature was contributed by Marc Mueller(PR 15512).

More Flexible TypedDict Creation and Update

Mypy was previously overly strict when type checking TypedDict creation and update operations. Though these checks were often technically correct, they sometimes triggered for apparently valid code. These checks have now been relaxed by default. You can enable stricter checking by using the new –extra-checks flag.

Construction using the ** syntax is now more flexible:

```
from typing import TypedDict

class A(TypedDict):
    foo: int
    bar: int

class B(TypedDict):
    foo: int

a: A = {"foo": 1, "bar": 2}
b: B = {"foo": 3}
a2: A = { **a, **b}  # OK (previously an error)
```

You can also call update() with a TypedDict argument that contains a subset of the keys in the updated TypedDict:

```
a.update(b)  # OK (previously an error)
```

This feature was contributed by Ivan Levkivskyi (PR 15425).

Deprecated Flag: –strict-concatenate

The behavior of –strict-concatenate is now included in the new –extra-checks flag, and the old flag is deprecated.

Optionally Show Links to Error Code Documentation

If you use –show-error-code-links, mypy will add documentation links to (many) reported errors. The links are not shown for error messages that are sufficiently obvious, and they are shown once per error code only.

Example output:

```
a.py:1: error: Need type annotation for "foo" (hint: "x: List[<type>] = ...")  [var-
↪annotated]
a.py:1: note: See https://mypy.rtfd.io/en/stable/_refs.html#code-var-annotated for more␣
↪info
```

This was contributed by Ivan Levkivskyi (PR 15449).

Consistently Avoid Type Checking Unreachable Code

If a module top level has unreachable code, mypy won't type check the unreachable statements. This is consistent with how functions behave. The behavior of –warn-unreachable is also more consistent now.

This was contributed by Ilya Priven (PR 15386).

Experimental Improved Type Inference for Generic Functions

You can use –new-type-inference to opt into an experimental new type inference algorithm. It fixes issues when calling a generic functions with an argument that is also a generic function, in particular. This current implementation is still incomplete, but we encourage trying it out and reporting bugs if you encounter regressions. We are planning to enable the new algorithm by default in a future mypy release.

This feature was contributed by Ivan Levkivskyi (PR 15287).

Partial Support for Python 3.12

Mypy and mypyc now support running on recent Python 3.12 development versions. Not all new Python 3.12 features are supported, and we don't ship compiled wheels for Python 3.12 yet.

- Fix ast warnings for Python 3.12 (Nikita Sobolev, PR 15558)
- mypyc: Fix multiple inheritance with a protocol on Python 3.12 (Jukka Lehtosalo, PR 15572)
- mypyc: Fix self-compilation on Python 3.12 (Jukka Lehtosalo, PR 15582)
- mypyc: Fix 3.12 issue with pickling of instances with __dict__ (Jukka Lehtosalo, PR 15574)
- mypyc: Fix i16 on Python 3.12 (Jukka Lehtosalo, PR 15510)
- mypyc: Fix int operations on Python 3.12 (Jukka Lehtosalo, PR 15470)
- mypyc: Fix generators on Python 3.12 (Jukka Lehtosalo, PR 15472)
- mypyc: Fix classes with __dict__ on 3.12 (Jukka Lehtosalo, PR 15471)
- mypyc: Fix coroutines on Python 3.12 (Jukka Lehtosalo, PR 15469)

- mypyc: Don't use _PyErr_ChainExceptions on 3.12, since it's deprecated (Jukka Lehtosalo, PR 15468)
- mypyc: Add Python 3.12 feature macro (Jukka Lehtosalo, PR 15465)

Improvements to Dataclasses

- Improve signature of dataclasses.replace (Ilya Priven, PR 14849)
- Fix dataclass/protocol crash on joining types (Ilya Priven, PR 15629)
- Fix strict optional handling in dataclasses (Ivan Levkivskyi, PR 15571)
- Support optional types for custom dataclass descriptors (Marc Mueller, PR 15628)
- Add `__slots__` attribute to dataclasses (Nikita Sobolev, PR 15649)
- Support better __post_init__ method signature for dataclasses (Nikita Sobolev, PR 15503)

Mypyc Improvements

- Support unsigned 8-bit native integer type: mypy_extensions.u8 (Jukka Lehtosalo, PR 15564)
- Support signed 16-bit native integer type: mypy_extensions.i16 (Jukka Lehtosalo, PR 15464)
- Define mypy_extensions.i16 in stubs (Jukka Lehtosalo, PR 15562)
- Document more unsupported features and update supported features (Richard Si, PR 15524)
- Fix final NamedTuple classes (Richard Si, PR 15513)
- Use C99 compound literals for undefined tuple values (Jukka Lehtosalo, PR 15453)
- Don't explicitly assign NULL values in setup functions (Logan Hunt, PR 15379)

Stubgen Improvements

- Teach stubgen to work with complex and unary expressions (Nikita Sobolev, PR 15661)
- Support ParamSpec and TypeVarTuple (Ali Hamdan, PR 15626)
- Fix crash on non-str docstring (Ali Hamdan, PR 15623)

Documentation Updates

- Add documentation for additional error codes (Ivan Levkivskyi, PR 15539)
- Improve documentation of type narrowing (Ilya Priven, PR 15652)
- Small improvements to protocol documentation (Shantanu, PR 15460)
- Remove confusing instance variable example in cheat sheet (Adel Atallah, PR 15441)

Other Notable Fixes and Improvements

- Constant fold additional unary and binary expressions (Richard Si, PR 15202)
- Exclude the same special attributes from Protocol as CPython (Kyle Benesch, PR 15490)
- Change the default value of the slots argument of attrs.define to True, to match runtime behavior (Ilya Priven, PR 15642)
- Fix type of class attribute if attribute is defined in both class and metaclass (Alex Waygood, PR 14988)
- Handle type the same as typing.Type in the first argument of classmethods (Erik Kemperman, PR 15297)
- Fix –find-occurrences flag (Shantanu, PR 15528)

- Fix error location for class patterns (Nikita Sobolev, PR 15506)
- Fix re-added file with errors in mypy daemon (Ivan Levkivskyi, PR 15440)
- Fix dmypy run on Windows (Ivan Levkivskyi, PR 15429)
- Fix abstract and non-abstract variant error for property deleter (Shantanu, PR 15395)
- Remove special casing for "cannot" in error messages (Ilya Priven, PR 15428)
- Add runtime `__slots__` attribute to attrs classes (Nikita Sobolev, PR 15651)
- Add get_expression_type to CheckerPluginInterface (Ilya Priven, PR 15369)
- Remove parameters that no longer exist from NamedTuple._make() (Alex Waygood, PR 15578)
- Allow using typing.Self in `__all__` with an explicit @staticmethod decorator (Erik Kemperman, PR 15353)
- Fix self types in subclass methods without Self annotation (Ivan Levkivskyi, PR 15541)
- Check for abstract class objects in tuples (Nikita Sobolev, PR 15366)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Adel Atallah
- Alex Waygood
- Ali Hamdan
- Erik Kemperman
- Federico Padua
- Ilya Priven
- Ivan Levkivskyi
- Jelle Zijlstra
- Jared Hance
- Jukka Lehtosalo
- Kyle Benesch
- Logan Hunt
- Marc Mueller
- Nikita Sobolev
- Richard Si
- Shantanu
- Stavros Ntentos
- Valentin Stanciu

Posted by Valentin Stanciu

1.36.20 Mypy 1.4

Tuesday, 20 June 2023

We've just uploaded mypy 1.4 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

The Override Decorator

Mypy can now ensure that when renaming a method, overrides are also renamed. You can explicitly mark a method as overriding a base class method by using the @typing.override decorator (PEP 698). If the method is then renamed in the base class while the method override is not, mypy will generate an error. The decorator will be available in typing in Python 3.12, but you can also use the backport from a recent version of `typing_extensions` on all supported Python versions.

This feature was contributed byThomas M Kehrenberg (PR 14609).

Propagating Type Narrowing to Nested Functions

Previously, type narrowing was not propagated to nested functions because it would not be sound if the narrowed variable changed between the definition of the nested function and the call site. Mypy will now propagate the narrowed type if the variable is not assigned to after the definition of the nested function:

```
def outer(x: str | None = None) -> None:
    if x is None:
        x = calculate_default()
    reveal_type(x)  # "str" (narrowed)

    def nested() -> None:
        reveal_type(x)  # Now "str" (used to be "str | None")

    nested()
```

This may generate some new errors because asserts that were previously necessary may become tautological or no-ops.

This was contributed by Jukka Lehtosalo (PR 15133).

Narrowing Enum Values Using "=="

Mypy now allows narrowing enum types using the == operator. Previously this was only supported when using the is operator. This makes exhaustiveness checking with enum types more usable, as the requirement to use the is operator was not very intuitive. In this example mypy can detect that the developer forgot to handle the value MyEnum.C in example

```
from enum import Enum

class MyEnum(Enum):
    A = 0
    B = 1
    C = 2

def example(e: MyEnum) -> str:  # Error: Missing return statement
    if e == MyEnum.A:
```

(continues on next page)

(continued from previous page)

```
        return 'x'
    elif e == MyEnum.B:
        return 'y'
```

Adding an extra elif case resolves the error:

```
...
def example(e: MyEnum) -> str:  # No error -- all values covered
    if e == MyEnum.A:
        return 'x'
    elif e == MyEnum.B:
        return 'y'
    elif e == MyEnum.C:
        return 'z'
```

This change can cause false positives in test cases that have assert statements like assert o.x == SomeEnum.X when using –strict-equality. Example:

```
# mypy: strict-equality

from enum import Enum

class MyEnum(Enum):
    A = 0
    B = 1

class C:
    x: MyEnum
    ...

def test_something() -> None:
    c = C(...)
    assert c.x == MyEnum.A
    c.do_something_that_changes_x()
    assert c.x == MyEnum.B  # Error: Non-overlapping equality check
```

These errors can be ignored using # type: ignore[comparison-overlap], or you can perform the assertion using a temporary variable as a workaround:

```
...
def test_something() -> None:
    ...
    x = c.x
    assert x == MyEnum.A  # Does not narrow c.x
    c.do_something_that_changes_x()
    x = c.x
    assert x == MyEnum.B  # OK
```

This feature was contributed by Shantanu (PR 11521).

Performance Improvements

- Speed up simplification of large union types and also fix a recursive tuple crash (Shantanu, PR 15128)
- Speed up union subtyping (Shantanu, PR 15104)
- Don't type check most function bodies when type checking third-party library code, or generally when ignoring errors (Jukka Lehtosalo, PR 14150)

Improvements to Plugins

- attrs.evolve: Support generics and unions (Ilya Konstantinov, PR 15050)
- Fix ctypes plugin (Alex Waygood)

Fixes to Crashes

- Fix a crash when function-scope recursive alias appears as upper bound (Ivan Levkivskyi, PR 15159)
- Fix crash on follow_imports_for_stubs (Ivan Levkivskyi, PR 15407)
- Fix stubtest crash in explicit init subclass (Shantanu, PR 15399)
- Fix crash when indexing TypedDict with empty key (Shantanu, PR 15392)
- Fix crash on NamedTuple as attribute (Ivan Levkivskyi, PR 15404)
- Correctly track loop depth for nested functions/classes (Ivan Levkivskyi, PR 15403)
- Fix crash on joins with recursive tuples (Ivan Levkivskyi, PR 15402)
- Fix crash with custom ErrorCode subclasses (Marc Mueller, PR 15327)
- Fix crash in dataclass protocol with self attribute assignment (Ivan Levkivskyi, PR 15157)
- Fix crash on lambda in generic context with generic method in body (Ivan Levkivskyi, PR 15155)
- Fix recursive type alias crash in make_simplified_union (Ivan Levkivskyi, PR 15216)

Improvements to Error Messages

- Use lower-case built-in collection types such as list[. . .] instead of List[. . .] in errors when targeting Python 3.9+ (Max Murin, PR 15070)
- Use X | Y union syntax in error messages when targeting Python 3.10+ (Omar Silva, PR 15102)
- Use type instead of Type in errors when targeting Python 3.9+ (Rohit Sanjay, PR 15139)
- Do not show unused-ignore errors in unreachable code, and make it a real error code (Ivan Levkivskyi, PR 15164)
- Don't limit the number of errors shown by default (Rohit Sanjay, PR 15138)
- Improver message for truthy functions (madt2709, PR 15193)
- Output distinct types when type names are ambiguous (teresa0605, PR 15184)
- Update message about invalid exception type in try (AJ Rasmussen, PR 15131)
- Add explanation if argument type is incompatible because of an unsupported numbers type (Jukka Lehtosalo, PR 15137)
- Add more detail to 'signature incompatible with supertype' messages for non-callables (Ilya Priven, PR 15263)

Documentation Updates

- Add –local-partial-types note to dmypy docs (Alan Du, PR 15259)
- Update getting started docs for mypyc for Windows (Valentin Stanciu, PR 15233)
- Clarify usage of callables regarding type object in docs (Viicos, PR 15079)
- Clarify difference between disallow_untyped_defs and disallow_incomplete_defs (Ilya Priven, PR 15247)
- Use attrs and @attrs.define in documentation and tests (Ilya Priven, PR 15152)

Mypyc Improvements

- Fix unexpected TypeError for certain variables with an inferred optional type (Richard Si, PR 15206)
- Inline math literals (Logan Hunt, PR 15324)
- Support unpacking mappings in dict display (Richard Si, PR 15203)

Changes to Stubgen

- Do not remove Generic from base classes (Ali Hamdan, PR 15316)
- Support yield from statements (Ali Hamdan, PR 15271)
- Fix missing total from TypedDict class (Ali Hamdan, PR 15208)
- Fix call-based namedtuple omitted from class bases (Ali Hamdan, PR 14680)
- Support TypedDict alternative syntax (Ali Hamdan, PR 14682)
- Make stubgen respect MYPY_CACHE_DIR (Henrik Bäärnhielm, PR 14722)
- Fixes and simplifications (Ali Hamdan, PR 15232)

Other Notable Fixes and Improvements

- Fix nested async functions when using TypeVar value restriction (Jukka Lehtosalo, PR 14705)
- Always allow returning Any from lambda (Ivan Levkivskyi, PR 15413)
- Add foundation for TypeVar defaults (PEP 696) (Marc Mueller, PR 14872)
- Update semantic analyzer for TypeVar defaults (PEP 696) (Marc Mueller, PR 14873)
- Make dict expression inference more consistent (Ivan Levkivskyi, PR 15174)
- Do not block on duplicate base classes (Nikita Sobolev, PR 15367)
- Generate an error when both staticmethod and classmethod decorators are used (Juhi Chandalia, PR 15118)
- Fix assert_type behaviour with literals (Carl Karsten, PR 15123)
- Fix match subject ignoring redefinitions (Vincent Vanlaer, PR 15306)
- Support `__all__`.remove (Shantanu, PR 15279)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Adrian Garcia Badaracco
- AJ Rasmussen
- Alan Du
- Alex Waygood
- Ali Hamdan
- Carl Karsten
- dosisod
- Ethan Smith
- Gregory Santosa
- Heather White
- Henrik Bäärnhielm
- Ilya Konstantinov
- Ilya Priven
- Ivan Levkivskyi
- Juhi Chandalia
- Jukka Lehtosalo
- Logan Hunt
- madt2709
- Marc Mueller
- Max Murin
- Nikita Sobolev
- Omar Silva
- Özgür
- Richard Si
- Rohit Sanjay
- Shantanu
- teresa0605
- Thomas M Kehrenberg
- Tin Tvrtković
- Tushar Sadhwani
- Valentin Stanciu
- Viicos
- Vincent Vanlaer
- Wesley Collin Wright

- William Santosa
- yaegassy

I'd also like to thank my employer, Dropbox, for supporting mypy development.

Posted by Jared Hance

1.36.21 Mypy 1.3

Wednesday, 10 May 2023

We've just uploaded mypy 1.3 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Performance Improvements

- Improve performance of union subtyping (Shantanu, PR 15104)
- Add negative subtype caches (Ivan Levkivskyi, PR 14884)

Stub Tooling Improvements

- Stubtest: Check that the stub is abstract if the runtime is, even when the stub is an overloaded method (Alex Waygood, PR 14955)
- Stubtest: Verify stub methods or properties are decorated with @final if they are decorated with @final at runtime (Alex Waygood, PR 14951)
- Stubtest: Fix stubtest false positives with TypedDicts at runtime (Alex Waygood, PR 14984)
- Stubgen: Support @functools.cached_property (Nikita Sobolev, PR 14981)
- Improvements to stubgenc (Chad Dombrova, PR 14564)

Improvements to attrs

- Add support for converters with TypeVars on generic attrs classes (Chad Dombrova, PR 14908)
- Fix attrs.evolve on bound TypeVar (Ilya Konstantinov, PR 15022)

Documentation Updates

- Improve async documentation (Shantanu, PR 14973)
- Improvements to cheat sheet (Shantanu, PR 14972)
- Add documentation for bytes formatting error code (Shantanu, PR 14971)
- Convert insecure links to use HTTPS (Marti Raudsepp, PR 14974)
- Also mention overloads in async iterator documentation (Shantanu, PR 14998)
- stubtest: Improve allowlist documentation (Shantanu, PR 15008)
- Clarify "Using types… but not at runtime" (Jon Shea, PR 15029)
- Fix alignment of cheat sheet example (Ondřej Cvacho, PR 15039)
- Fix error for callback protocol matching against callable type object (Shantanu, PR 15042)

Error Reporting Improvements

- Improve bytes formatting error (Shantanu, PR 14959)

Mypyc Improvements

- Fix unions of bools and ints (Tomer Chachamu, PR 15066)

Other Fixes and Improvements

- Fix narrowing union types that include Self with isinstance (Christoph Tyralla, PR 14923)
- Allow objects matching SupportsKeysAndGetItem to be unpacked (Bryan Forbes, PR 14990)
- Check type guard validity for staticmethods (EXPLOSION, PR 14953)
- Fix sys.platform when cross-compiling with emscripten (Ethan Smith, PR 14888)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alex Waygood
- Amin Alaee
- Bryan Forbes
- Chad Dombrova
- Charlie Denton
- Christoph Tyralla
- dosisod
- Ethan Smith
- EXPLOSION
- Ilya Konstantinov
- Ivan Levkivskyi
- Jon Shea
- Jukka Lehtosalo
- KotlinIsland
- Marti Raudsepp
- Nikita Sobolev
- Ondřej Cvacho
- Shantanu
- sobolevn
- Tomer Chachamu
- Yaroslav Halchenko

Posted by Wesley Collin Wright.

1.36.22 Mypy 1.2

Thursday, 6 April 2023

We've just uploaded mypy 1.2 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Improvements to Dataclass Transforms

- Support implicit default for "init" parameter in field specifiers (Wesley Collin Wright and Jukka Lehtosalo, PR 15010)
- Support descriptors in dataclass transform (Jukka Lehtosalo, PR 15006)
- Fix frozen_default in incremental mode (Wesley Collin Wright)
- Fix frozen behavior for base classes with direct metaclasses (Wesley Collin Wright, PR 14878)

Mypyc: Native Floats

Mypyc now uses a native, unboxed representation for values of type float. Previously these were heap-allocated Python objects. Native floats are faster and use less memory. Code that uses floating-point operations heavily can be several times faster when using native floats.

Various float operations and math functions also now have optimized implementations. Refer to the documentation for a full list.

This can change the behavior of existing code that uses subclasses of float. When assigning an instance of a subclass of float to a variable with the float type, it gets implicitly converted to a float instance when compiled:

```
from lib import MyFloat  # MyFloat ia a subclass of "float"

def example() -> None:
    x = MyFloat(1.5)
    y: float = x  # Implicit conversion from MyFloat to float
    print(type(y))  # float, not MyFloat
```

Previously, implicit conversions were applied to int subclasses but not float subclasses.

Also, int values can no longer be assigned to a variable with type float in compiled code, since these types now have incompatible representations. An explicit conversion is required:

```
def example(n: int) -> None:
    a: float = 1  # Error: cannot assign "int" to "float"
    b: float = 1.0  # OK
    c: float = n  # Error
    d: float = float(n)  # OK
```

This restriction only applies to assignments, since they could otherwise narrow down the type of a variable from float to int. int values can still be implicitly converted to float when passed as arguments to functions that expect float values.

Note that mypyc still doesn't support arrays of unboxed float values. Using list[float] involves heap-allocated float objects, since list can only store boxed values. Support for efficient floating point arrays is one of the next major planned mypyc features.

Related changes:

- Use a native unboxed representation for floats (Jukka Lehtosalo, PR 14880)
- Document native floats and integers (Jukka Lehtosalo, PR 14927)
- Fixes to float to int conversion (Jukka Lehtosalo, PR 14936)

Mypyc: Native Integers

Mypyc now supports signed 32-bit and 64-bit integer types in addition to the arbitrary-precision int type. You can use the types mypy_extensions.i32 and mypy_extensions.i64 to speed up code that uses integer operations heavily.

Simple example:

```
from mypy_extensions import i64

def inc(x: i64) -> i64:
    return x + 1
```

Refer to the documentation for more information. This feature was contributed by Jukka Lehtosalo.

Other Mypyc Fixes and Improvements

- Support iterating over a TypedDict (Richard Si, PR 14747)
- Faster coercions between different tuple types (Jukka Lehtosalo, PR 14899)
- Faster calls via type aliases (Jukka Lehtosalo, PR 14784)
- Faster classmethod calls via cls (Jukka Lehtosalo, PR 14789)

Fixes to Crashes

- Fix crash on class-level import in protocol definition (Ivan Levkivskyi, PR 14926)
- Fix crash on single item union of alias (Ivan Levkivskyi, PR 14876)
- Fix crash on ParamSpec in incremental mode (Ivan Levkivskyi, PR 14885)

Documentation Updates

- Update adopting –strict documentation for 1.0 (Shantanu, PR 14865)
- Some minor documentation tweaks (Jukka Lehtosalo, PR 14847)
- Improve documentation of top level mypy: disable-error-code comment (Nikita Sobolev, PR 14810)

Error Reporting Improvements

- Add error code to `typing_extensions` suggestion (Shantanu, PR 14881)
- Add a separate error code for top-level await (Nikita Sobolev, PR 14801)
- Don't suggest two obsolete stub packages (Jelle Zijlstra, PR 14842)
- Add suggestions for pandas-stubs and lxml-stubs (Shantanu, PR 14737)

Other Fixes and Improvements

- Multiple inheritance considers callable objects as subtypes of functions (Christoph Tyralla, PR 14855)
- stubtest: Respect @final runtime decorator and enforce it in stubs (Nikita Sobolev, PR 14922)
- Fix false positives related to type[<type-var>] (sterliakov, PR 14756)
- Fix duplication of ParamSpec prefixes and properly substitute ParamSpecs (EXPLOSION, PR 14677)
- Fix line number if `__iter__` is incorrectly reported as missing (Jukka Lehtosalo, PR 14893)
- Fix incompatible overrides of overloaded generics with self types (Shantanu, PR 14882)
- Allow SupportsIndex in slice expressions (Shantanu, PR 14738)
- Support if statements in bodies of dataclasses and classes that use dataclass_transform (Jacek Chałupka, PR 14854)
- Allow iterable class objects to be unpacked (including enums) (Alex Waygood, PR 14827)
- Fix narrowing for walrus expressions used in match statements (Shantanu, PR 14844)
- Add signature for attr.evolve (Ilya Konstantinov, PR 14526)
- Fix Any inference when unpacking iterators that don't directly inherit from typing.Iterator (Alex Waygood, PR 14821)
- Fix unpack with overloaded `__iter__` method (Nikita Sobolev, PR 14817)
- Reduce size of JSON data in mypy cache (dosisod, PR 14808)
- Improve "used before definition" checks when a local definition has the same name as a global definition (Stas Ilinskiy, PR 14517)
- Honor NoReturn as __setitem__ return type to mark unreachable code (sterliakov, PR 12572)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alex Waygood
- Avasam
- Christoph Tyralla
- dosisod
- EXPLOSION
- Ilya Konstantinov
- Ivan Levkivskyi
- Jacek Chałupka
- Jelle Zijlstra
- Jukka Lehtosalo
- Marc Mueller

- Max Murin
- Nikita Sobolev
- Richard Si
- Shantanu
- Stas Ilinskiy
- sterliakov
- Wesley Collin Wright

Posted by Jukka Lehtosalo

1.36.23 Mypy 1.1.1

Monday, 6 March 2023

We've just uploaded mypy 1.1.1 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

Support for `dataclass_transform``

This release adds full support for the dataclass_transform decorator defined in PEP 681. This allows decorators, base classes, and metaclasses that generate a __init__ method or other methods based on the properties of that class (similar to dataclasses) to have those methods recognized by mypy.

This was contributed by Wesley Collin Wright.

Dedicated Error Code for Method Assignments

Mypy can't safely check all assignments to methods (a form of monkey patching), so mypy generates an error by default. To make it easier to ignore this error, mypy now uses the new error code method-assign for this. By disabling this error code in a file or globally, mypy will no longer complain about assignments to methods if the signatures are compatible.

Mypy also supports the old error code assignment for these assignments to prevent a backward compatibility break. More generally, we can use this mechanism in the future if we wish to split or rename another existing error code without causing backward compatibility issues.

This was contributed by Ivan Levkivskyi (PR 14570).

Fixes to Crashes

- Fix a crash on walrus in comprehension at class scope (Ivan Levkivskyi, PR 14556)
- Fix crash related to value-constrained TypeVar (Shantanu, PR 14642)

Fixes to Cache Corruption

- Fix generic TypedDict/NamedTuple caching (Ivan Levkivskyi, PR 14675)

Mypyc Fixes and Improvements

- Raise "non-trait base must be first…" error less frequently (Richard Si, PR 14468)
- Generate faster code for bool comparisons and arithmetic (Jukka Lehtosalo, PR 14489)
- Optimize __(a)enter__/__(a)exit__ for native classes (Jared Hance, PR 14530)
- Detect if attribute definition conflicts with base class/trait (Jukka Lehtosalo, PR 14535)
- Support __(r)divmod__ dunders (Richard Si, PR 14613)
- Support __pow__, __rpow__, and __ipow__ dunders (Richard Si, PR 14616)
- Fix crash on star unpacking to underscore (Ivan Levkivskyi, PR 14624)
- Fix iterating over a union of dicts (Richard Si, PR 14713)

Fixes to Detecting Undefined Names (used-before-def)

- Correctly handle walrus operator (Stas Ilinskiy, PR 14646)
- Handle walrus declaration in match subject correctly (Stas Ilinskiy, PR 14665)

Stubgen Improvements

Stubgen is a tool for automatically generating draft stubs for libraries.

- Allow aliases below the top level (Chad Dombrova, PR 14388)
- Fix crash with PEP 604 union in type variable bound (Shantanu, PR 14557)
- Preserve PEP 604 unions in generated .pyi files (hamdanal, PR 14601)

Stubtest Improvements

Stubtest is a tool for testing that stubs conform to the implementations.

- Update message format so that it's easier to go to error location (Avasam, PR 14437)
- Handle name-mangling edge cases better (Alex Waygood, PR 14596)

Changes to Error Reporting and Messages

- Add new TypedDict error code typeddict-unknown-key (JoaquimEsteves, PR 14225)
- Give arguments a more reasonable location in error messages (Max Murin, PR 14562)
- In error messages, quote just the module's name (Ilya Konstantinov, PR 14567)
- Improve misleading message about Enum() (Rodrigo Silva, PR 14590)
- Suggest importing from `typing_extensions` if definition is not in typing (Shantanu, PR 14591)
- Consistently use type-abstract error code (Ivan Levkivskyi, PR 14619)
- Consistently use literal-required error code for TypedDicts (Ivan Levkivskyi, PR 14621)
- Adjust inconsistent dataclasses plugin error messages (Wesley Collin Wright, PR 14637)
- Consolidate literal bool argument error messages (Wesley Collin Wright, PR 14693)

Other Fixes and Improvements

- Check that type guards accept a positional argument (EXPLOSION, PR 14238)
- Fix bug with in operator used with a union of Container and Iterable (Max Murin, PR 14384)
- Support protocol inference for type[T] via metaclass (Ivan Levkivskyi, PR 14554)
- Allow overlapping comparisons between bytes-like types (Shantanu, PR 14658)
- Fix mypy daemon documentation link in README (Ivan Levkivskyi, PR 14644)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alex Waygood
- Avasam
- Chad Dombrova
- dosisod
- EXPLOSION
- hamdanal
- Ilya Konstantinov
- Ivan Levkivskyi
- Jared Hance
- JoaquimEsteves
- Jukka Lehtosalo
- Marc Mueller
- Max Murin
- Michael Lee
- Michael R. Crusoe
- Richard Si
- Rodrigo Silva
- Shantanu
- Stas Ilinskiy
- Wesley Collin Wright
- Yilei "Dolee" Yang
- Yurii Karabas

We'd also like to thank our employer, Dropbox, for funding the mypy core team.

Posted by Max Murin

1.36.24 Mypy 1.0

Monday, 6 February 2023

We've just uploaded mypy 1.0 to the Python Package Index (PyPI). Mypy is a static type checker for Python. This release includes new features, performance improvements and bug fixes. You can install it as follows:

```
python3 -m pip install -U mypy
```

You can read the full documentation for this release on Read the Docs.

New Release Versioning Scheme

Now that mypy reached 1.0, we'll switch to a new versioning scheme. Mypy version numbers will be of form x.y.z.

Rules:

- The major release number (x) is incremented if a feature release includes a significant backward incompatible change that affects a significant fraction of users.
- The minor release number (y) is incremented on each feature release. Minor releases include updated stdlib stubs from typeshed.
- The point release number (z) is incremented when there are fixes only.

Mypy doesn't use SemVer, since most minor releases have at least minor backward incompatible changes in typeshed, at the very least. Also, many type checking features find new legitimate issues in code. These are not considered backward incompatible changes, unless the number of new errors is very high.

Any significant backward incompatible change must be announced in the blog post for the previous feature release, before making the change. The previous release must also provide a flag to explicitly enable or disable the new behavior (whenever practical), so that users will be able to prepare for the changes and report issues. We should keep the feature flag for at least a few releases after we've switched the default.

See "Release Process" in the mypy wiki for more details and for the most up-to-date version of the versioning scheme.

Performance Improvements

Mypy 1.0 is up to 40% faster than mypy 0.991 when type checking the Dropbox internal codebase. We also set up a daily job to measure the performance of the most recent development version of mypy to make it easier to track changes in performance.

Many optimizations contributed to this improvement:

- Improve performance for errors on class with many attributes (Shantanu, PR 14379)
- Speed up make_simplified_union (Jukka Lehtosalo, PR 14370)
- Micro-optimize get_proper_type(s) (Jukka Lehtosalo, PR 14369)
- Micro-optimize flatten_nested_unions (Jukka Lehtosalo, PR 14368)
- Some semantic analyzer micro-optimizations (Jukka Lehtosalo, PR 14367)
- A few miscellaneous micro-optimizations (Jukka Lehtosalo, PR 14366)
- Optimization: Avoid a few uses of contextmanagers in semantic analyzer (Jukka Lehtosalo, PR 14360)
- Optimization: Enable always defined attributes in Type subclasses (Jukka Lehtosalo, PR 14356)
- Optimization: Remove expensive context manager in type analyzer (Jukka Lehtosalo, PR 14357)
- subtypes: fast path for Union/Union subtype check (Hugues, PR 14277)
- Micro-optimization: avoid Bogus[int] types that cause needless boxing (Jukka Lehtosalo, PR 14354)

- Avoid slow error message logic if errors not shown to user (Jukka Lehtosalo, PR 14336)
- Speed up the implementation of hasattr() checks (Jukka Lehtosalo, PR 14333)
- Avoid the use of a context manager in hot code path (Jukka Lehtosalo, PR 14331)
- Change various type queries into faster bool type queries (Jukka Lehtosalo, PR 14330)
- Speed up recursive type check (Jukka Lehtosalo, PR 14326)
- Optimize subtype checking by avoiding a nested function (Jukka Lehtosalo, PR 14325)
- Optimize type parameter checks in subtype checking (Jukka Lehtosalo, PR 14324)
- Speed up freshening type variables (Jukka Lehtosalo, PR 14323)
- Optimize implementation of TypedDict types for **kwds (Jukka Lehtosalo, PR 14316)

Warn About Variables Used Before Definition

Mypy will now generate an error if you use a variable before it's defined. This feature is enabled by default. By default mypy reports an error when it infers that a variable is always undefined.

```
y = x  # E: Name "x" is used before definition [used-before-def]
x = 0
```

This feature was contributed by Stas Ilinskiy.

Detect Possibly Undefined Variables (Experimental)

A new experimental possibly-undefined error code is now available that will detect variables that may be undefined:

```
    if b:
        x = 0
    print(x)  # Error: Name "x" may be undefined [possibly-undefined]
```

The error code is disabled be default, since it can generate false positives.

This feature was contributed by Stas Ilinskiy.

Support the "Self" Type

There is now a simpler syntax for declaring generic self types introduced in PEP 673: the Self type. You no longer have to define a type variable to use "self types", and you can use them with attributes. Example from mypy documentation:

```
from typing import Self

class Friend:
    other: Self | None = None

    @classmethod
    def make_pair(cls) -> tuple[Self, Self]:
        a, b = cls(), cls()
        a.other = b
        b.other = a
        return a, b

class SuperFriend(Friend):
    pass
```

(continues on next page)

(continued from previous page)

```
# a and b have the inferred type "SuperFriend", not "Friend"
a, b = SuperFriend.make_pair()
```

The feature was introduced in Python 3.11. In earlier Python versions a backport of Self is available in `typing_extensions`.

This was contributed by Ivan Levkivskyi (PR 14041).

Support ParamSpec in Type Aliases

ParamSpec and Concatenate can now be used in type aliases. Example:

```
from typing import ParamSpec, Callable

P = ParamSpec("P")
A = Callable[P, None]

def f(c: A[int, str]) -> None:
    c(1, "x")
```

This feature was contributed by Ivan Levkivskyi (PR 14159).

ParamSpec and Generic Self Types No Longer Experimental

Support for ParamSpec (PEP 612) and generic self types are no longer considered experimental.

Miscellaneous New Features

- Minimal, partial implementation of dataclass_transform (PEP 681) (Wesley Collin Wright, PR 14523)
- Add basic support for `typing_extensions`.TypeVar (Marc Mueller, PR 14313)
- Add –debug-serialize option (Marc Mueller, PR 14155)
- Constant fold initializers of final variables (Jukka Lehtosalo, PR 14283)
- Enable Final instance attributes for attrs (Tin Tvrtković, PR 14232)
- Allow function arguments as base classes (Ivan Levkivskyi, PR 14135)
- Allow super() with mixin protocols (Ivan Levkivskyi, PR 14082)
- Add type inference for dict.keys membership (Matthew Hughes, PR 13372)
- Generate error for class attribute access if attribute is defined with `__slots__` (Harrison McCarty, PR 14125)
- Support additional attributes in callback protocols (Ivan Levkivskyi, PR 14084)

Fixes to Crashes

- Fix crash on prefixed ParamSpec with forward reference (Ivan Levkivskyi, PR 14569)
- Fix internal crash when resolving the same partial type twice (Shantanu, PR 14552)
- Fix crash in daemon mode on new import cycle (Ivan Levkivskyi, PR 14508)
- Fix crash in mypy daemon (Ivan Levkivskyi, PR 14497)
- Fix crash on Any metaclass in incremental mode (Ivan Levkivskyi, PR 14495)
- Fix crash in await inside comprehension outside function (Ivan Levkivskyi, PR 14486)

- Fix crash in Self type on forward reference in upper bound (Ivan Levkivskyi, PR 14206)
- Fix a crash when incorrect super() is used outside a method (Ivan Levkivskyi, PR 14208)
- Fix crash on overriding with frozen attrs (Ivan Levkivskyi, PR 14186)
- Fix incremental mode crash on generic function appearing in nested position (Ivan Levkivskyi, PR 14148)
- Fix daemon crash on malformed NamedTuple (Ivan Levkivskyi, PR 14119)
- Fix crash during ParamSpec inference (Ivan Levkivskyi, PR 14118)
- Fix crash on nested generic callable (Ivan Levkivskyi, PR 14093)
- Fix crashes with unpacking SyntaxError (Shantanu, PR 11499)
- Fix crash on partial type inference within a lambda (Ivan Levkivskyi, PR 14087)
- Fix crash with enums (Michael Lee, PR 14021)
- Fix crash with malformed TypedDicts and disllow-any-expr (Michael Lee, PR 13963)

Error Reporting Improvements

- More helpful error for missing self (Shantanu, PR 14386)
- Add error-code truthy-iterable (Marc Mueller, PR 13762)
- Fix pluralization in error messages (KotlinIsland, PR 14411)

Mypyc: Support Match Statement

Mypyc can now compile Python 3.10 match statements.

This was contributed by dosisod (PR 13953).

Other Mypyc Fixes and Improvements

- Optimize int(x)/float(x)/complex(x) on instances of native classes (Richard Si, PR 14450)
- Always emit warnings (Richard Si, PR 14451)
- Faster bool and integer conversions (Jukka Lehtosalo, PR 14422)
- Support attributes that override properties (Jukka Lehtosalo, PR 14377)
- Precompute set literals for "in" operations and iteration (Richard Si, PR 14409)
- Don't load targets with forward references while setting up non-extension class `__all__` (Richard Si, PR 14401)
- Compile away NewType type calls (Richard Si, PR 14398)
- Improve error message for multiple inheritance (Joshua Bronson, PR 14344)
- Simplify union types (Jukka Lehtosalo, PR 14363)
- Fixes to union simplification (Jukka Lehtosalo, PR 14364)
- Fix for typeshed changes to Collection (Shantanu, PR 13994)
- Allow use of enum.Enum (Shantanu, PR 13995)
- Fix compiling on Arch Linux (dosisod, PR 13978)

Documentation Improvements

- Various documentation and error message tweaks (Jukka Lehtosalo, PR 14574)
- Improve Generics documentation (Shantanu, PR 14587)
- Improve protocols documentation (Shantanu, PR 14577)
- Improve dynamic typing documentation (Shantanu, PR 14576)
- Improve the Common Issues page (Shantanu, PR 14581)
- Add a top-level TypedDict page (Shantanu, PR 14584)
- More improvements to getting started documentation (Shantanu, PR 14572)
- Move truthy-function documentation from "optional checks" to "enabled by default" (Anders Kaseorg, PR 14380)
- Avoid use of implicit optional in decorator factory documentation (Tom Schraitle, PR 14156)
- Clarify documentation surrounding install-types (Shantanu, PR 14003)
- Improve searchability for module level type ignore errors (Shantanu, PR 14342)
- Advertise mypy daemon in README (Ivan Levkivskyi, PR 14248)
- Add link to error codes in README (Ivan Levkivskyi, PR 14249)
- Document that report generation disables cache (Ilya Konstantinov, PR 14402)
- Stop saying mypy is beta software (Ivan Levkivskyi, PR 14251)
- Flycheck-mypy is deprecated, since its functionality was merged to Flycheck (Ivan Levkivskyi, PR 14247)
- Update code example in "Declaring decorators" (ChristianWitzler, PR 14131)

Stubtest Improvements

Stubtest is a tool for testing that stubs conform to the implementations.

- Improve error message for `__all__`-related errors (Alex Waygood, PR 14362)
- Improve heuristics for determining whether global-namespace names are imported (Alex Waygood, PR 14270)
- Catch BaseException on module imports (Shantanu, PR 14284)
- Associate exported symbol error with `__all__` object_path (Nikita Sobolev, PR 14217)
- Add __warningregistry__ to the list of ignored module dunders (Nikita Sobolev, PR 14218)
- If a default is present in the stub, check that it is correct (Jelle Zijlstra, PR 14085)

Stubgen Improvements

Stubgen is a tool for automatically generating draft stubs for libraries.

- Treat dlls as C modules (Shantanu, PR 14503)

Other Notable Fixes and Improvements

- Update stub suggestions based on recent typeshed changes (Alex Waygood, PR 14265)
- Fix attrs protocol check with cache (Marc Mueller, PR 14558)
- Fix strict equality check if operand item type has custom __eq__ (Jukka Lehtosalo, PR 14513)
- Don't consider object always truthy (Jukka Lehtosalo, PR 14510)

- Properly support union of TypedDicts as dict literal context (Ivan Levkivskyi, PR 14505)
- Properly expand type in generic class with Self and TypeVar with values (Ivan Levkivskyi, PR 14491)
- Fix recursive TypedDicts/NamedTuples defined with call syntax (Ivan Levkivskyi, PR 14488)
- Fix type inference issue when a class inherits from Any (Shantanu, PR 14404)
- Fix false positive on generic base class with six (Ivan Levkivskyi, PR 14478)
- Don't read scripts without extensions as modules in namespace mode (Tim Geypens, PR 14335)
- Fix inference for constrained type variables within unions (Christoph Tyralla, PR 14396)
- Fix Unpack imported from typing (Marc Mueller, PR 14378)
- Allow trailing commas in ini configuration of multiline values (Nikita Sobolev, PR 14240)
- Fix false negatives involving Unions and generators or coroutines (Shantanu, PR 14224)
- Fix ParamSpec constraint for types as callable (Vincent Vanlaer, PR 14153)
- Fix type aliases with fixed-length tuples (Jukka Lehtosalo, PR 14184)
- Fix issues with type aliases and new style unions (Jukka Lehtosalo, PR 14181)
- Simplify unions less aggressively (Ivan Levkivskyi, PR 14178)
- Simplify callable overlap logic (Ivan Levkivskyi, PR 14174)
- Try empty context when assigning to union typed variables (Ivan Levkivskyi, PR 14151)
- Improvements to recursive types (Ivan Levkivskyi, PR 14147)
- Make non-numeric non-empty FORCE_COLOR truthy (Shantanu, PR 14140)
- Fix to recursive type aliases (Ivan Levkivskyi, PR 14136)
- Correctly handle Enum name on Python 3.11 (Ivan Levkivskyi, PR 14133)
- Fix class objects falling back to metaclass for callback protocol (Ivan Levkivskyi, PR 14121)
- Correctly support self types in callable ClassVar (Ivan Levkivskyi, PR 14115)
- Fix type variable clash in nested positions and in attributes (Ivan Levkivskyi, PR 14095)
- Allow class variable as implementation for read only attribute (Ivan Levkivskyi, PR 14081)
- Prevent warnings from causing dmypy to fail (Andrzej Bartosiński, PR 14102)
- Correctly process nested definitions in mypy daemon (Ivan Levkivskyi, PR 14104)
- Don't consider a branch unreachable if there is a possible promotion (Ivan Levkivskyi, PR 14077)
- Fix incompatible overrides of overloaded methods in concrete subclasses (Shantanu, PR 14017)
- Fix new style union syntax in type aliases (Jukka Lehtosalo, PR 14008)
- Fix and optimise overload compatibility checking (Shantanu, PR 14018)
- Improve handling of redefinitions through imports (Shantanu, PR 13969)
- Preserve (some) implicitly exported types (Shantanu, PR 13967)

Typeshed Updates

Typeshed is now modular and distributed as separate PyPI packages for everything except the standard library stubs. Please see git log for full list of typeshed changes.

Acknowledgements

Thanks to all mypy contributors who contributed to this release:

- Alessio Izzo
- Alex Waygood
- Anders Kaseorg
- Andrzej Bartosiński
- Avasam
- ChristianWitzler
- Christoph Tyralla
- dosisod
- Harrison McCarty
- Hugo van Kemenade
- Hugues
- Ilya Konstantinov
- Ivan Levkivskyi
- Jelle Zijlstra
- jhance
- johnthagen
- Jonathan Daniel
- Joshua Bronson
- Jukka Lehtosalo
- KotlinIsland
- Lakshay Bisht
- Lefteris Karapetsas
- Marc Mueller
- Matthew Hughes
- Michael Lee
- Nick Drozd
- Nikita Sobolev
- Richard Si
- Shantanu
- Stas Ilinskiy
- Tim Geypens
- Tin Tvrtković
- Tom Schraitle
- Valentin Stanciu

- Vincent Vanlaer

We'd also like to thank our employer, Dropbox, for funding the mypy core team.

Posted by Stas Ilinskiy

1.36.25 Previous releases

For information about previous releases, refer to the posts at https://mypy-lang.blogspot.com/

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

C

D

E

F

H

I

J

L

M

N

P

R

S

T

U

V

W

X