Chapter 8. Type Hints in Functions

> *It should also be emphasized that* **Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.**
>
> —Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa, PEP 484—Type Hints[1]

Type hints are the biggest change in the history of Python since the unification of types and classes in Python 2.2, released in 2001. However, type hints do not benefit all Python users equally. That's why they should always be optional.

PEP 484—Type Hints introduced syntax and semantics for explicit type declarations in function arguments, return values, and variables. The goal is to help developer tools find bugs in Python codebases via static analysis, i.e., without actually running the code through tests.

The main beneficiaries are professional software engineers using IDEs (Integrated Development Environments) and CI (Continuous Integration). The cost-benefit analysis that makes type hints attractive to that group does not apply to all users of Python.

Python's user base is much wider than that. It includes scientists, traders, journalists, artists, makers, analysts, and students in many fields—among others. For most of them, the cost of learning type hints is likely higher—unless they already know a language with static types, subtyping, and generics. The benefits will be lower for many of those users, given how they interact with Python, and the smaller size of their codebases and teams—often "teams" of one. Python's default dynamic typing is simpler and more expressive when writing code for exploring data and ideas, as in data science, creative computing, and learning,

This chapter focuses on Python's type hints in function signatures. Chapter 15 explores type hints in the context of classes, and other `typing` module features.

The major topics in this chapter are:

- A hands-on introduction to gradual typing with Mypy
- The complementary perspectives of duck typing and nominal typing
- Overview of the main categories of types that can appear in annotations—this is about 60% of the chapter
- Type hinting variadic parameters (`*args` , `**kwargs`)
- Limitations and downsides of type hints and static typing

What's New in This Chapter

This chapter is completely new. Type hints appeared in Python 3.5 after I wrapped up the first edition of *Fluent Python*.

Given the limitations of a static type system, the best idea of PEP 484 was to introduce a *gradual type system*. Let's begin by defining what that means.

About Gradual Typing

PEP 484 introduced a *gradual type system* to Python. Other languages with gradual type systems are Microsoft's TypeScript, Dart (the language of the Flutter SDK, created by Google), and Hack (a dialect of PHP supported by Facebook's HHVM virtual machine). The Mypy type checker itself started as a language: a gradually typed dialect of Python with its own interpreter. Guido van Rossum convinced the creator of Mypy, Jukka Lehtosalo, to make it a tool for checking annotated Python code.

A gradual type system:

Is optional

By default, the type checker should not emit warnings for code that has no type hints. Instead, the type checker assumes the `Any` type when it cannot determine the type of an object. The `Any` type is considered compatible with all other types.

Does not catch type errors at runtime

Type hints are used by static type checkers, linters, and IDEs to raise warnings. They do not prevent inconsistent values from being passed to functions or assigned to variables at runtime.

Does not enhance performance

> Type annotations provide data that could, in theory, allow optimizations in the generated bytecode, but such optimizations are not implemented in any Python runtime that I am aware in of July 2021.[2]

The best usability feature of gradual typing is that annotations are always optional.

With static type systems, most type constraints are easy to express, many are cumbersome, some are hard, and a few are impossible.[3] You may very well write an excellent piece of Python code, with good test coverage and passing tests, but still be unable to add type hints that satisfy a type checker. That's OK; just leave out the problematic type hints and ship it!

Type hints are optional at all levels: you can have entire packages with no type hints, you can silence the type checker when you import one of those packages into a module where you use type hints, and you can add special comments to make the type checker ignore specific lines in your code.

TIP

Seeking 100% coverage of type hints is likely to stimulate type hinting without proper thought, only to satisfy the metric. It will also prevent teams from making the most of the power and flexibility of Python. Code without type hints should naturally be accepted when annotations would make an API less user-friendly, or unduly complicate its implementation.

Gradual Typing in Practice

Let's see how gradual typing works in practice, starting with a simple function and gradually adding type hints to it, guided by Mypy.

NOTE

There are several Python type checkers compatible with PEP 484, including Google's pytype, Microsoft's Pyright, Facebook's Pyre—in addition to type checkers embedded in IDEs such as PyCharm. I picked Mypy for the examples because it's the best known. However, one of the others may be a better fit for some projects or teams. Pytype, for example, is designed to handle codebases with no type hints and still provide useful advice. It is more lenient than Mypy, and can also generate annotations for your code.

We will annotate a `show_count` function that returns a string with a count and a singular or plural word, depending on the count:

```
>>> show_count(99, 'bird')
'99 birds'
>>> show_count(1, 'bird')
'1 bird'
>>> show_count(0, 'bird')
'no birds'
```

Example 8-1 shows the source code of `show_count`, without annotations.

Example 8-1. `show_count` **from *messages.py* without type hints**

```
def show_count(count, word):
    if count == 1:
        return f'1 {word}'
    count_str = str(count) if count else 'no'
    return f'{count_str} {word}s'
```

Starting with Mypy

To begin type checking, I run the `mypy` command on the *messages.py* module:

```
…/no_hints/ $ pip install mypy
[lots of messages omitted...]
…/no_hints/ $ mypy messages.py
Success: no issues found in 1 source file
```

Mypy with default settings finds no problem with Example 8-1.

WARNING

I am using Mypy 0.910, the most recent release as I review this in July 2021. The Mypy "Introduction" warns that it "is officially beta software. There will be occasional changes that break backward compatibility." Mypy is giving me at least one report that is not the same I got when I wrote this chapter in April 2020. By the time you read this, you may get different results than shown here.

If a function signature has no annotations, Mypy ignores it by default—unless configured otherwise.

For Example 8-2, I also have `pytest` unit tests. This is the code in *messages_test.py*.

Example 8-2. *messages_test.py* without type hints

```
from pytest import mark

from messages import show_count

@mark.parametrize('qty, expected', [
    (1, '1 part'),
    (2, '2 parts'),
])
def test_show_count(qty, expected):
    got = show_count(qty, 'part')
    assert got == expected

def test_show_count_zero():
    got = show_count(0, 'part')
    assert got == 'no parts'
```

Now let's add type hints, guided by Mypy.

Making Mypy More Strict

The command-line option `--disallow-untyped-defs` makes Mypy flag any function definition that does not have type hints for all its parameters and for its return value.

Using `--disallow-untyped-defs` on the test file produces three errors and a note:

```
…/no_hints/ $ mypy --disallow-untyped-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
messages_test.py:10: error: Function is missing a type annotation
messages_test.py:15: error: Function is missing a return type annotation
messages_test.py:15: note: Use "-> None" if function does not return a valu
Found 3 errors in 2 files (checked 1 source file)
```

For the first steps with gradual typing, I prefer to use another option: `--disallow-incomplete-defs`. Initially, it tells me nothing:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
Success: no issues found in 1 source file
```

Now I can add just the return type to `show_count` in *messages.py*:

```
def show_count(count, word) -> str:
```

This is enough to make Mypy look at it. Using the same command line as before to check *messages_test.py* will lead Mypy to look at *messages.py* again:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
for one or more arguments
Found 1 error in 1 file (checked 1 source file)
```

Now I can gradually add type hints function by function, without getting warnings about functions that I haven't annotated. This is a fully annotated signature that satisfies Mypy:

```
def show_count(count: int, word: str) -> str:
```

TIP

Instead of typing command-line options like `--disallow-incomplete-defs`, you can save your favorite as described in the Mypy configuration file documentation. You can have global settings and per-module settings. Here is a simple *mypy.ini* to get started:

```
[mypy]
python_version = 3.9
warn_unused_configs = True
disallow_incomplete_defs = True
```

A Default Parameter Value

The `show_count` function in Example 8-1 only works with regular nouns. If the plural can't be spelled by appending an `'s'`, we should let the user provide the plural form, like this:

```
>>> show_count(3, 'mouse', 'mice')
'3 mice'
```

Let's do a little "type-driven development." First we add a test that uses that third argument. Don't forget to add the return type hint to the test function, otherwise Mypy will not check it.

```
def test_irregular() -> None:
    got = show_count(2, 'child', 'children')
    assert got == '2 children'
```

Mypy detects the error:

```
…/hints_2/ $ mypy messages_test.py
messages_test.py:22: error: Too many arguments for "show_count"
Found 1 error in 1 file (checked 1 source file)
```

Now I edit `show_count`, adding the optional `plural` parameter in Example 8-3.

Example 8-3. `showcount` **from *hints_2/messages.py* with an optional parameter**

```
def show_count(count: int, singular: str, plural: str = '') -> str:
    if count == 1:
        return f'1 {singular}'
    count_str = str(count) if count else 'no'
    if not plural:
        plural = singular + 's'
    return f'{count_str} {plural}'
```

Now Mypy reports “Success.”

WARNING

Here is one typing mistake that Python does not catch. Can you spot it?

```
def hex2rgb(color=str) -> tuple[int, int, int]:
```

Mypy’s error report is not very helpful:

```
colors.py:24: error: Function is missing a type
    annotation for one or more arguments
```

The type hint for the `color` argument should be `color: str`. I wrote `color=str`, which is not an annotation: it sets the default value of `color` to `str`.

In my experience, it’s a common mistake and easy to overlook, especially in complicated type hints.

The following details are considered good style for type hints:

- No space between the parameter name and the `:`; one space after the `:`
- Spaces on both sides of the `=` that precedes a default parameter value

On the other hand, PEP 8 says there should be no spaces around the `=` if there is no type hint for that particular parameter.

CODE STYLE: USE FLAKE8 AND BLUE

Instead of memorizing such silly rules, use tools like *flake8* and *blue*. *flake8* reports on code styling, among many other issues, and *blue* rewrites source code according to (most) rules embedded in the *black* code formatting tool.

Given the goal of enforcing a "standard" coding style, *blue* is better than *black* because it follows Python's own style of using single quotes by default, double quotes as an alternative:

```
>>> "I prefer single quotes"
'I prefer single quotes'
```

The preference for single quotes is embedded in `repr()`, among other places in CPython. The *doctest* module depends on `repr()` using single quotes by default.

One of the authors of *blue* is Barry Warsaw, coauthor of PEP 8, Python core developer since 1994, and a member of Python's Steering Council from 2019 to present (July 2021). We are in very good company when we choose single quotes by default.

If you must use *black*, use the `black -S` option. Then it will leave your quotes as they are.

Using None as a Default

In Example 8-3, the parameter `plural` is annotated as `str`, and the default value is `''`, so there is no type conflict.

I like that solution, but in other contexts `None` is a better default. If the optional parameter expects a mutable type, then `None` is the only sensible default—as we saw in "Mutable Types as Parameter Defaults: Bad Idea".

To have `None` as the default for the `plural` parameter, here is what the signature would look like:

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] = None) ->
```

Let's unpack that:

- `Optional[str]` means `plural` may be a `str` or `None`.
- You must explicitly provide the default value `= None`.

If you don't assign a default value to `plural`, the Python runtime will treat it as a required parameter. Remember: at runtime, type hints are ignored.

Note that we need to import `Optional` from the `typing` module. When importing types, it's good practice to use the syntax `from typing import X` to reduce the length of the function signatures.

WARNING

`Optional` is not a great name, because that annotation does not make the parameter optional. What makes it optional is assigning a default value to the parameter. `Optional[str]` just means: the type of this parameter may be `str` or `NoneType`. In the Haskell and Elm languages, a similar type is named `Maybe`.

Now that we've had a first practical view of gradual typing, let's consider what the concept of *type* means in practice.

Types Are Defined by Supported Operations

> *There are many definitions of the concept of type in the literature. Here we assume that type is a set of values and a set of functions that one can apply to these values.*
>
> —PEP 483—The Theory of Type Hints

In practice, it's more useful to consider the set of supported operations as the defining characteristic of a type.[4]

For example, from the point of view of applicable operations, what are the valid types for `x` in the following function?

```
def double(x):
    return x * 2
```

The `x` parameter type may be numeric (`int`, `complex`, `Fraction`, `numpy.uint32`, etc.) but it may also be a sequence (`str`, `tuple`, `list`, `array`), an N-dimensional `numpy.array`, or any other type that implements or inherits a `__mul__` method that accepts an `int` argument.

However, consider this annotated `double`. Please ignore the missing return type for now, let's focus on the parameter type:

```
from collections import abc

def double(x: abc.Sequence):
    return x * 2
```

A type checker will reject that code. If you tell Mypy that `x` is of type `abc.Sequence`, it will flag `x * 2` as an error because the `Sequence` ABC does not implement or inherit the `__mul__` method. At runtime, that code will work with concrete sequences such as `str`, `tuple`, `list`, `array`, etc., as well as numbers, because at runtime the type hints are ignored. But the type checker only cares about what is explicitly declared, and `abc.Sequence` has no `__mul__`.

That's why the title of this section is "Types Are Defined by Supported Operations." The Python runtime accepts any object as the `x` argument for both versions of the `double` function. The computation `x * 2` may work, or it may raise `TypeError` if the operation is not supported by `x`. In contrast, Mypy will declare `x * 2` as wrong while analyzing the annotated `double` source code, because it's an unsupported operation for the declared type: `x: abc.Sequence`.

In a gradual type system, we have the interplay of two different views of types:

Duck typing

The view adopted by Smalltalk—the pioneering object-oriented language—as well as Python, JavaScript, and Ruby. Objects have types, but variables (including parameters) are untyped. In practice, it doesn't matter what the declared type of the object is, only what operations it actually supports. If I can invoke `birdie.quack()`, then `birdie` is a duck in this context. By definition, duck typing is only enforced at runtime, when operations on objects are attempted. This is more flexible than *nominal typing*, at the cost of allowing more errors at runtime.[5]

Nominal typing

The view adopted by C++, Java, and C#, supported by annotated Python. Objects and variables have types. But objects only exist at runtime, and the type checker only cares about the source code where variables (including parameters) are annotated with type hints. If `Duck` is a subclass of `Bird`, you can assign a `Duck` instance to a parameter annotated as `birdie: Bird`. But in the body of the function, the type checker considers the call `birdie.quack()` illegal, because `birdie` is nominally a `Bird`, and that class does not provide the `.quack()` method. It doesn't matter if the actual argument at runtime is a `Duck`, because nominal typing is enforced statically. The type checker doesn't run any part of the program, it only reads the source code. This is more rigid than *duck typing*, with the advantage of catching some bugs earlier in a build pipeline, or even as the code is typed in an IDE.

Example 8-4 is a silly example that contrasts duck typing and nominal typing, as well as static type checking and runtime behavior.[6]

Example 8-4. ***birds.py***

```
class Bird:
    pass

class Duck(Bird):  ❶
    def quack(self):
        print('Quack!')

def alert(birdie):  ❷
    birdie.quack()

def alert_duck(birdie: Duck) -> None:  ❸
```

```
    birdie.quack()

def alert_bird(birdie: Bird) -> None:  ❹
    birdie.quack()
```

❶ `Duck` is a subclass of `Bird`.

❷ `alert` has no type hints, so the type checker ignores it.

❸ `alert_duck` takes one argument of type `Duck`.

❹ `alert_bird` takes one argument of type `Bird`.

Type checking *birds.py* with Mypy, we see a problem:

```
…/birds/ $ mypy birds.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

Just by analyzing the source code, Mypy sees that `alert_bird` is problematic: the type hint declares the `birdie` parameter with type `Bird`, but the body of the function calls `birdie.quack()`—and the `Bird` class has no such method.

Now let's try to use the `birds` module in *daffy.py* in Example 8-5.

Example 8-5. *daffy.py*

```
from birds import *

daffy = Duck()
alert(daffy)        ❶
alert_duck(daffy)   ❷
alert_bird(daffy)   ❸
```

❶ Valid call, because `alert` has no type hints.

❷ Valid call, because `alert_duck` takes a `Duck` argument, and `daffy` is a `Duck`.

❸ Valid call, because `alert_bird` takes a `Bird` argument, and `daffy` is also a `Bird`—the superclass of `Duck`.

Running Mypy on *daffy.py* raises the same error about the `quack` call in the `alert_bird` function defined in *birds.py*:

```
…/birds/ $ mypy daffy.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

But Mypy sees no problem with *daffy.py* itself: the three function calls are OK.

Now, if you run *daffy.py*, this is what you get:

```
…/birds/ $ python3 daffy.py
Quack!
Quack!
Quack!
```

Everything works! Duck typing FTW!

At runtime, Python doesn't care about declared types. It uses duck typing only. Mypy flagged an error in `alert_bird`, but calling it with `daffy` works fine at runtime. This may surprise many Pythonistas at first: a static type checker will sometimes find errors in programs that we know will execute.

However, if months from now you are tasked with extending the silly bird example, you may be grateful for Mypy. Consider this *woody.py* module, which also uses `birds`, in Example 8-6.

Example 8-6. *woody.py*

```
from birds import *

woody = Bird()
alert(woody)
alert_duck(woody)
alert_bird(woody)
```

Mypy finds two errors while checking *woody.py*:

```
…/birds/ $ mypy woody.py
birds.py:16: error: "Bird" has no attribute "quack"
woody.py:5: error: Argument 1 to "alert_duck" has incompatible type "Bird";
expected "Duck"
Found 2 errors in 2 files (checked 1 source file)
```

The first error is in *birds.py*: the `birdie.quack()` call in `alert_bird`, which we've seen before. The second error is in *woody.py*: `woody` is an instance of `Bird`, so the call `alert_duck(woody)` is invalid because that function requires a `Duck`. Every `Duck` is a `Bird`, but not every `Bird` is a `Duck`.

At runtime, none of the calls in *woody.py* succeed. The succession of failures is best illustrated in a console session with callouts in Example 8-7.

Example 8-7. Runtime errors and how Mypy could have helped

```
>>> from birds import *
>>> woody = Bird()
>>> alert(woody)  ❶
Traceback (most recent call last):
  ...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_duck(woody) ❷
Traceback (most recent call last):
  ...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_bird(woody)  ❸
Traceback (most recent call last):
  ...
AttributeError: 'Bird' object has no attribute 'quack'
```

❶ Mypy could not detect this error because there are no type hints in `alert`.

❷ Mypy reported the problem: `Argument 1 to "alert_duck" has incompatible type "Bird"; expected "Duck"`.

❸ Mypy has been telling us since Example 8-4 that the body of the `alert_bird` function is wrong: `"Bird" has no attribute "quack"`.

This little experiment shows that duck typing is easier to get started and is more flexible, but allows unsupported operations to cause errors at runtime. Nominal typing detects errors before runtime, but sometimes can reject code that actually runs—such as the call `alert_bird(daffy)` in Example 8-5. Even if it sometimes works, the `alert_bird` function is misnamed: its body does require an object that supports the `.quack()` method, which `Bird` doesn't have.

In this silly example, the functions are one-liners. But in real code they could be longer; they could pass the `birdie` argument to more functions, and the origin of the `birdie` argument could be many function calls away, making it hard to pinpoint the cause of a runtime error. The type checker prevents many such errors from ever happening at runtime.

NOTE

The value of type hints is questionable in the tiny examples that fit in a book. The benefits grow with the size of the codebase. That's why companies with millions of lines of Python code—like Dropbox, Google, and Facebook—invested in teams and tools to support the company-wide adoption of type hints, and have significant and increasing portions of their Python codebases type checked in their CI pipelines.

In this section we explored the relationship of types and operations in duck typing and nominal typing, starting with the simple `double()` function—which we left without proper type hints. Now we will tour the most important types used for annotating functions. We'll see a good way to add type hints to `double()` when we reach "Static Protocols". But before we get to that, there are more fundamental types to know.

Types Usable in Annotations

Pretty much any Python type can be used in type hints, but there are restrictions and recommendations. In addition, the `typing` module introduced special constructs with semantics that are sometimes surprising.

This section covers all the major types you can use with annotations:

- `typing.Any`
- Simple types and classes

- `typing.Optional` and `typing.Union`
- Generic collections, including tuples and mappings
- Abstract base classes
- Generic iterables
- Parameterized generics and `TypeVar`
- `typing.Protocols` —the key to *static duck typing*
- `typing.Callable`
- `typing.NoReturn` —a good way to end this list

We'll cover each of these in turn, starting with a type that is strange, apparently useless, but crucially important.

The Any Type

The keystone of any gradual type system is the `Any` type, also known as the *dynamic type*. When a type checker sees an untyped function like this:

```
def double(x):
    return x * 2
```

it assumes this:

```
def double(x: Any) -> Any:
    return x * 2
```

That means the `x` argument and the return value can be of any type, including different types. `Any` is assumed to support every possible operation.

Contrast `Any` with `object`. Consider this signature:

```
def double(x: object) -> object:
```

This function also accepts arguments of every type, because every type is a *subtype-of* `object`.

However, a type checker will reject this function:

```
def double(x: object) -> object:
    return x * 2
```

The problem is that `object` does not support the `__mul__` operation. This is what Mypy reports:

```
…/birds/ $ mypy double_object.py
double_object.py:2: error: Unsupported operand types for * ("object" and "i
Found 1 error in 1 file (checked 1 source file)
```

More general types have narrower interfaces, i.e., they support fewer operations. The `object` class implements fewer operations than `abc.Sequence`, which implements fewer operations than `abc.MutableSequence`, which implements fewer operations than `list`.

But `Any` is a magic type that sits at the top and the bottom of the type hierarchy. It's simultaneously the most general type—so that an argument `n: Any` accepts values of every type—and the most specialized type, supporting every possible operation. At least, that's how the type checker understands `Any`.

Of course, no type can support every possible operation, so using `Any` prevents the type checker from fulfilling its core mission: detecting potentially illegal operations before your program crashes with a runtime exception.

Subtype-of versus consistent-with

Traditional object-oriented nominal type systems rely on the is *subtype-of* relationship. Given a class `T1` and a subclass `T2`, then `T2` is *subtype-of* `T1`.

Consider this code:

```
class T1:
    ...

class T2(T1):
    ...

def f1(p: T1) -> None:
    ...

o2 = T2()
```

```
f1(o2)  # OK
```

The call `f1(o2)` is an application of the Liskov Substitution Principle—LSP. Barbara Liskov[7] actually defined *is subtype-of* in terms of supported operations: if an object of type `T2` substitutes an object of type `T1` and the program still behaves correctly, then `T2` is *subtype-of* `T1`.

Continuing from the previous code, this shows a violation of the LSP:

```
def f2(p: T2) -> None:
    ...

o1 = T1()

f2(o1)  # type error
```

From the point of view of supported operations, this makes perfect sense: as a subclass, `T2` inherits and must support all operations that `T1` does. So an instance of `T2` can be used anywhere an instance of `T1` is expected. But the reverse is not necessarily true: `T2` may implement additional methods, so an instance of `T1` may not be used everywhere an instance of `T2` is expected. This focus on supported operations is reflected in the name *behavioral subtyping*, also used to refer to the LSP.

In a gradual type system, there is another relationship: *consistent-with*, which applies wherever *subtype-of* applies, with special provisions for type `Any`.

The rules for *consistent-with* are:

1. Given `T1` and a subtype `T2`, then `T2` is *consistent-with* `T1` (Liskov substitution).
2. Every type is *consistent-with* `Any`: you can pass objects of every type to an argument declared of type `Any`.
3. `Any` is *consistent-with* every type: you can always pass an object of type `Any` where an argument of another type is expected.

Considering the previous definitions of the objects `o1` and `o2`, here are examples of valid code, illustrating rules #2 and #3:

```
def f3(p: Any) -> None:
    ...

o0 = object()
o1 = T1()
o2 = T2()

f3(o0)  #
f3(o1)  #  all OK: rule #2
f3(o2)  #

def f4():  # implicit return type: `Any`
    ...

o4 = f4()  # inferred type: `Any`

f1(o4)  #
f2(o4)  #  all OK: rule #3
f3(o4)  #
```

Every gradual type system needs a wildcard type like `Any`.

TIP

The verb "to infer" is a fancy synomym for "to guess," used in the context of type analysis. Modern type checkers in Python and other languages don't require type annotations everywhere because they can infer the type of many expressions. For example, if I write `x = len(s) * 10`, the type checker doesn't need an explicit local declaration to know that `x` is an `int`, as long as it can find type hints for the `len` built-in.

Now we can explore the rest of the types used in annotations.

Simple Types and Classes

Simple types like `int`, `float`, `str`, and `bytes` may be used directly in type hints. Concrete classes from the standard library, external packages, or user defined— `FrenchDeck`, `Vector2d`, and `Duck` —may also be used in type hints.

Abstract base classes are also useful in type hints. We'll get back to them as we study collection types, and in "Abstract Base Classes".

Among classes, *consistent-with* is defined like *subtype-of*: a subclass is *consistent-with* all its superclasses.

However, "practicality beats purity," so there is an important exception, which I discuss in the following tip.

INT IS CONSISTENT-WITH COMPLEX

There is no nominal subtype relationship between the built-in types `int`, `float`, and `complex`: they are direct subclasses of `object`. But PEP 484 declares that `int` is *consistent-with* `float`, and `float` is *consistent-with* `complex`. It makes sense in practice: `int` implements all operations that `float` does, and `int` implements additional ones as well—bitwise operations like `&`, `|`, `<<`, etc. The end result is: `int` is *consistent-with* `complex`. For `i = 3`, `i.real` is `3`, and `i.imag` is `0`.

Optional and Union Types

We saw the `Optional` special type in "Using None as a Default". It solves the problem of having `None` as a default, as in this example from that section:

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] = None) ->
```

The construct `Optional[str]` is actually a shortcut for `Union[str, None]`, which means the type of `plural` may be `str` or `None`.

BETTER SYNTAX FOR OPTIONAL AND UNION IN PYTHON 3.10

We can write `str | bytes` instead of `Union[str, bytes]` since Python 3.10. It's less typing, and there's no need to import `Optional` or `Union` from `typing`. Contrast the old and new syntax for the type hint of the `plural` parameter of `show_count`:

```
plural: Optional[str] = None     # before
plural: str | None = None        # after
```

The `|` operator also works with `isinstance` and `issubclass` to build the second argument: `isinstance(x, int | str)`. For more, see PEP 604—Complementary syntax for Union[].

The `ord` built-in function's signature is a simple example of `Union`—it accepts `str` or `bytes`, and returns an `int`:[8]

```
def ord(c: Union[str, bytes]) -> int: ...
```

Here is an example of a function that takes a `str`, but may return a `str` or a `float`:

```
from typing import Union

def parse_token(token: str) -> Union[str, float]:
    try:
        return float(token)
    except ValueError:
        return token
```

If possible, avoid creating functions that return `Union` types, as they put an extra burden on the user—forcing them to check the type of the returned value at runtime to know what to do with it. But the `parse_token` in the preceding code is a reasonable use case in the context of a simple expression evaluator.

TIP

In "Dual-Mode str and bytes APIs", we saw functions that accept either `str` or `bytes` arguments, but return `str` if the argument was `str` or `bytes` if the arguments was `bytes`. In those cases, the return type is determined by the input type, so `Union` is not an accurate solution. To properly annotate such functions, we need a type variable—presented in "Parameterized Generics and TypeVar"—or overloading, which we'll see in "Overloaded Signatures".

`Union[]` requires at least two types. Nested `Union` types have the same effect as a flattened `Union`. So this type hint:

```
Union[A, B, Union[C, D, E]]
```

is the same as:

```
Union[A, B, C, D, E]
```

`Union` is more useful with types that are not consistent among themselves. For example: `Union[int, float]` is redundant because `int` is *consistent-with* `float`. If you just use `float` to annotate the parameter, it will accept `int` values as well.

Generic Collections

Most Python collections are heterogeneous. For example, you can put any mixture of different types in a `list`. However, in practice that's not very useful: if you put objects in a collection, you are likely to want to operate on them later, and usually this means they must share at least one common method.[9]

Generic types can be declared with type parameters to specify the type of the items they can handle.

For example, a `list` can be parameterized to constrain the type of the elements in it, as you can see in Example 8-8.

Example 8-8. `tokenize` with type hints for Python ≥ 3.9

```
def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

In Python ≥ 3.9, it means that `tokenize` returns a `list` where every item is of type `str`.

The annotations `stuff: list` and `stuff: list[Any]` mean the same thing: `stuff` is a list of objects of any type.

TIP

If you are using Python 3.8 or earlier, the concept is the same, but you need more code to make it work—as explained in the optional box "Legacy Support and Deprecated Collection Types".

PEP 585—Type Hinting Generics In Standard Collections lists collections from the standard library accepting generic type hints. The following list shows only those collections that use the simplest form of generic type hint, `container[item]`:

```
list        collections.deque        abc.Sequence    abc.MutableSequence
set         abc.Container            abc.Set         abc.MutableSet
frozenset   abc.Collection
```

The `tuple` and mapping types support more complex type hints, as we'll see in their respective sections.

As of Python 3.10, there is no good way to annotate `array.array`, taking into account the `typecode` constructor argument, which determines whether integers or floats are stored in the array. An even harder problem is how to type check integer ranges to prevent `OverflowError` at runtime when adding elements to arrays. For example, an `array` with `typecode='B'` can only hold `int` values from 0 to 255. Currently, Python's static type system is not up to this challenge.

LEGACY SUPPORT AND DEPRECATED COLLECTION TYPES

(You may skip this box if you only use Python 3.9 or later.)

For Python 3.7 and 3.8, you need a `__future__` import to make the `[]` notation work with built-in collections such as `list`, as shown in Example 8-9.

Example 8-9. `tokenize` **with type hints for Python ≥ 3.7**

```
from __future__ import annotations

def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

The `__future__` import does not work with Python 3.6 or earlier. Example 8-10 shows how to annotate `tokenize` in a way that works with Python ≥ 3.5.

Example 8-10. `tokenize` **with type hints for Python ≥ 3.5**

```
from typing import List

def tokenize(text: str) -> List[str]:
    return text.upper().split()
```

To provide the initial support for generic type hints, the authors of PEP 484 created dozens of generic types in the `typing` module. Table 8-1 shows some of them. For the full list, visit the *typing* documentation.

Table 8-1. Some collection types and their type hint equivalents

Collection	Type hint equivalent
`list`	`typing.List`
`set`	`typing.Set`
`frozenset`	`typing.FrozenSet`
`collections.deque`	`typing.Deque`
`collections.abc.MutableSequence`	`typing.MutableSequence`
`collections.abc.Sequence`	`typing.Sequence`
`collections.abc.Set`	`typing.AbstractSet`
`collections.abc.MutableSet`	`typing.MutableSet`

PEP 585—Type Hinting Generics In Standard Collections started a multi-year process to improve the usability of generic type hints. We can summarize that process in four steps:

1. Introduce `from __future__ import annotations` in Python 3.7 to enable the use of standard library classes as generics with `list[str]` notation.
2. Make that behavior the default in Python 3.9: `list[str]` now works without the `future` import.
3. Deprecate all the redundant generic types from the `typing` module.[10] Deprecation warnings will not be issued by the Python interpreter because type checkers should flag the deprecated types when the checked program targets Python 3.9 or newer.
4. Remove those redundant generic types in the first version of Python released five years after Python 3.9. At the current cadence, that could be Python 3.14, a.k.a Python Pi.

Now let's see how to annotate generic tuples.

Tuple Types

There are three ways to annotate tuple types:

- Tuples as records
- Tuples as records with named fields
- Tuples as immutable sequences

Tuples as records

If you're using a `tuple` as a record, use the `tuple` built-in and declare the types of the fields within `[]`.

For example, the type hint would be `tuple[str, float, str]` to accept a tuple with city name, population, and country: `('Shanghai', 24.28, 'China')`.

Consider a function that takes a pair of geographic coordinates and returns a Geohash, used like this:

```
>>> shanghai = 31.2304, 121.4737
>>> geohash(shanghai)
'wtw3sjq6q'
```

Example 8-11 shows how `geohash` is defined, using the `geolib` package from PyPI.

Example 8-11. *coordinates.py* with the `geohash` function

```
from geolib import geohash as gh  # type: ignore  ❶

PRECISION = 9

def geohash(lat_lon: tuple[float, float]) -> str:  ❷
    return gh.encode(*lat_lon, PRECISION)
```

❶ This comment stops Mypy from reporting that the `geolib` package doesn't have type hints.

❷ `lat_lon` parameter annotated as a `tuple` with two `float` fields.

TIP

For Python < 3.9, import and use `typing.Tuple` in type hints. It is deprecated but will remain in the standard library at least until 2024.

Tuples as records with named fields

To annotate a tuple with many fields, or specific types of tuple your code uses in many places, I highly recommend using `typing.NamedTuple`, as seen in Chapter 5. Example 8-12 shows a variation of Example 8-11 with `NamedTuple`.

Example 8-12. *coordinates_named.py* with the `NamedTuple` `Coordinates` and the `geohash` function

```
from typing import NamedTuple

from geolib import geohash as gh  # type: ignore

PRECISION = 9

class Coordinate(NamedTuple):
    lat: float
    lon: float

def geohash(lat_lon: Coordinate) -> str:
    return gh.encode(*lat_lon, PRECISION)
```

As explained in "Overview of Data Class Builders", `typing.NamedTuple` is a factory for `tuple` subclasses, so `Coordinate` is *consistent-with* `tuple[float, float]` but the reverse is not true—after all, `Coordinate` has extra methods added by `NamedTuple`, like `._asdict()`, and could also have user-defined methods.

In practice, this means that it is type safe to pass a `Coordinate` instance to the `display` function defined in the following:

```
def display(lat_lon: tuple[float, float]) -> str:
    lat, lon = lat_lon
    ns = 'N' if lat >= 0 else 'S'
    ew = 'E' if lon >= 0 else 'W'
    return f'{abs(lat):0.1f}°{ns}, {abs(lon):0.1f}°{ew}'
```

Tuples as immutable sequences

To annotate tuples of unspecified length that are used as immutable lists, you must specify a single type, followed by a comma and `...` (that's Python's ellipsis token, made of three periods, not Unicode `U+2026` — `HORIZONTAL ELLIPSIS`).

For example, `tuple[int, ...]` is a tuple with `int` items.

The ellipsis indicates that any number of elements >= 1 is acceptable. There is no way to specify fields of different types for tuples of arbitrary length.

The annotations `stuff: tuple[Any, ...]` and `stuff: tuple` mean the same thing: `stuff` is a tuple of unspecified length with objects of any type.

Here is a `columnize` function that transforms a sequence into a table of rows and cells in the form of a list of tuples with unspecified lengths. This is useful to display items in columns, like this:

```
>>> animals = 'drake fawn heron ibex koala lynx tahr xerus yak zapus'.split
>>> table = columnize(animals)
>>> table
[('drake', 'koala', 'yak'), ('fawn', 'lynx', 'zapus'), ('heron', 'tahr'),
 ('ibex', 'xerus')]
>>> for row in table:
...     print(''.join(f'{word:10}' for word in row))
...
drake     koala     yak
fawn      lynx      zapus
heron     tahr
ibex      xerus
```

Example 8-13 shows the implementation of `columnize`. Note the return type:

```
list[tuple[str, ...]]
```

Example 8-13. *columnize.py* returns a list of tuples of strings

```
from collections.abc import Sequence

def columnize(
    sequence: Sequence[str], num_columns: int = 0
) -> list[tuple[str, ...]]:
    if num_columns == 0:
        num_columns = round(len(sequence) ** 0.5)
    num_rows, reminder = divmod(len(sequence), num_columns)
    num_rows += bool(reminder)
    return [tuple(sequence[i::num_rows]) for i in range(num_rows)]
```

Generic Mappings

Generic mapping types are annotated as `MappingType[KeyType, ValueType]`. The built-in `dict` and the mapping types in `collections` and `collections.abc` accept that notation in Python ≥ 3.9. For earlier versions, you must use `typing.Dict` and other mapping types from the `typing` module, as described in "Legacy Support and Deprecated Collection Types".

Example 8-14 shows a practical use of a function returning an inverted index to search Unicode characters by name—a variation of Example 4-21 more suitable for server-side code that we'll study in Chapter 21.

Given starting and ending Unicode character codes, `name_index` returns a `dict[str, set[str]]`, which is an inverted index mapping each word to a set of characters that have that word in their names. For example, after indexing ASCII characters from 32 to 64, here are the sets of characters mapped to the words `'SIGN'` and `'DIGIT'`, and how to find the character named `'DIGIT EIGHT'`:

```
>>> index = name_index(32, 65)
>>> index['SIGN']
{'$', '>', '=', '+', '<', '%', '#'}
>>> index['DIGIT']
{'8', '5', '6', '2', '3', '0', '1', '4', '7', '9'}
>>> index['DIGIT'] & index['EIGHT']
{'8'}
```

Example 8-14 shows the source code for *charindex.py* with the `name_index` function. Besides a `dict[]` type hint, this example has three features appearing for the first time in the book.

Example 8-14. ***charindex.py***

```
import sys
import re
import unicodedata
from collections.abc import Iterator

RE_WORD = re.compile(r'\w+')
STOP_CODE = sys.maxunicode + 1

def tokenize(text: str) -> Iterator[str]:  ❶
    """return iterable of uppercased words"""
    for match in RE_WORD.finditer(text):
        yield match.group().upper()

def name_index(start: int = 32, end: int = STOP_CODE) -> dict[str, set[str]
    index: dict[str, set[str]] = {}  ❷
    for char in (chr(i) for i in range(start, end)):
        if name := unicodedata.name(char, ''):  ❸
            for word in tokenize(name):
                index.setdefault(word, set()).add(char)
    return index
```

❶ `tokenize` is a generator function. Chapter 17 is about generators.

❷ The local variable `index` is annotated. Without the hint, Mypy says: `Need type annotation for 'index' (hint: "index: dict[<type>, <type>] = ...")`.

❸ I used the walrus operator `:=` in the `if` condition. It assigns the result of the `unicodedata.name()` call to `name`, and the whole expression evaluates to that result. When the result is `''`, that's falsy, and the `index` is not updated.[11]

NOTE

When using a `dict` as a record, it is common to have all keys of the `str` type, with values of different types depending on the keys. That is covered in "TypedDict".

Abstract Base Classes

> *Be conservative in what you send, be liberal in what you accept.*
>
> —Postel's law, a.k.a. the Robustness Principle

Table 8-1 lists several abstract classes from `collections.abc`. Ideally, a function should accept arguments of those abstract types—or their `typing` equivalents before Python 3.9—and not concrete types. This gives more flexibility to the caller.

Consider this function signature:

```
from collections.abc import Mapping

def name2hex(name: str, color_map: Mapping[str, int]) -> str:
```

Using `abc.Mapping` allows the caller to provide an instance of `dict`, `defaultdict`, `ChainMap`, a `UserDict` subclass, or any other type that is a *subtype-of* `Mapping`.

In contrast, consider this signature:

```
def name2hex(name: str, color_map: dict[str, int]) -> str:
```

Now `color_map` must be a `dict` or one of its subtypes, such as `defaultDict` or `OrderedDict`. In particular, a subclass of `collections.UserDict` would not pass the type check for `color_map`, despite being the recommended way to create user-defined mappings, as we saw in "Subclassing UserDict Instead of dict". Mypy would reject a `UserDict` or an instance of a class derived from it, because `UserDict` is not a subclass of `dict`; they are siblings. Both are subclasses of `abc.MutableMapping`.[12]

Therefore, in general it's better to use `abc.Mapping` or `abc.MutableMapping` in parameter type hints, instead of `dict` (or `typing.Dict` in legacy code). If the `name2hex` function doesn't need to mutate the given `color_map`, the most accurate type hint for `color_map` is `abc.Mapping`. That way, the caller doesn't need to provide an object that implements methods like `setdefault`, `pop`, and `update`, which are part of the `MutableMapping` interface, but not of `Mapping`. This has to do with the second part of Postel's law: "Be liberal in what you accept."

Postel's law also tells us to be conservative in what we send. The return value of a function is always a concrete object, so the return type hint should be a concrete type, as in the example from "Generic Collections"—which uses `list[str]`:

```
def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

Under the entry of `typing.List`, the Python documentation says:

> *Generic version of `list`. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as `Sequence` or `Iterable`.*

A similar comment appears in the entries for `typing.Dict` and `typing.Set`.

Remember that most ABCs from `collections.abc` and other concrete classes from `collections`, as well as built-in collections, support generic type hint notation like `collections.deque[str]` starting with Python 3.9. The corresponding `typing` collections are only needed to support code written in Python 3.8 or earlier. The full list of classes that became generic appears in the "Implementation" section of PEP 585—Type Hinting Generics In Standard Collections.

To wrap up our discussion of ABCs in type hints, we need to talk about the `numbers` ABCs.

The fall of the numeric tower

The `numbers` package defines the so-called *numeric tower* described in PEP 3141—A Type Hierarchy for Numbers. The tower is linear hierarchy of ABCs, with `Number` at the top:

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

Those ABCs work perfectly well for runtime type checking, but they are not supported for static type checking. The "Numeric Tower" section of PEP 484 rejects the `numbers` ABCs and dictates that the built-in types `complex`, `float`, and `int` should be treated as special cases, as explained in "int Is Consistent-With complex".

We'll come back to this issue in "The numbers ABCs and Numeric Protocols", in Chapter 13, which is devoted to contrasting protocols and ABCs.

In practice, if you want to annotate numeric arguments for static type checking, you have a few options:

1. Use one of the concrete types `int`, `float`, or `complex`—as recommended by PEP 488.
2. Declare a union type like `Union[float, Decimal, Fraction]`.
3. If you want to avoid hardcoding concrete types, use numeric protocols like `SupportsFloat`, covered in "Runtime Checkable Static Protocols".

The upcoming section "Static Protocols" is a prerequisite for understanding the numeric protocols.

Meanwhile, let's get to one of the most useful ABCs for type hints: `Iterable`.

Iterable

The `typing.List` documentation I just quoted recommends `Sequence` and `Iterable` for function parameter type hints.

One example of the `Iterable` argument appears in the `math.fsum` function from the standard library:

```
def fsum(__seq: Iterable[float]) -> float:
```

STUB FILES AND THE TYPESHED PROJECT

As of Python 3.10, the standard library has no annotations, but Mypy, PyCharm, etc. can find the necessary type hints in the Typeshed project, in the form of *stub files*: special source files with a *.pyi* extension that have annotated function and method signatures, without the implementation—much like header files in C.

The signature for `math.fsum` is in */stdlib/2and3/math.pyi*. The leading underscores in `__seq` are a PEP 484 convention for positional-only parameters, explained in "Annotating Positional Only and Variadic Parameters".

Example 8-15 is another example using an `Iterable` parameter that produces items that are `tuple[str, str]`. Here is how the function is used:

```
>>> l33t = [('a', '4'), ('e', '3'), ('i', '1'), ('o', '0')]
>>> text = 'mad skilled noob powned leet'
>>> from replacer import zip_replace
>>> zip_replace(text, l33t)
'm4d sk1ll3d n00b p0wn3d l33t'
```

Example 8-15 shows how it's implemented.

Example 8-15. ***replacer.py***

```
from collections.abc import Iterable

FromTo = tuple[str, str]  ❶

def zip_replace(text: str, changes: Iterable[FromTo]) -> str:  ❷
    for from_, to in changes:
        text = text.replace(from_, to)
    return text
```

❶ `FromTo` is a *type alias*: I assigned `tuple[str, str]` to `FromTo`, to make the signature of `zip_replace` more readable.

❷ `changes` needs to be an `Iterable[FromTo]`; that's the same as `Iterable[tuple[str, str]]`, but shorter and easier to read.

EXPLICIT TYPEALIAS IN PYTHON 3.10

PEP 613—Explicit Type Aliases introduced a special type, `TypeAlias`, to make the assignments that create type aliases more visible and easier to type check. Starting with Python 3.10, this is the preferred way to create type aliases:

```
from typing import TypeAlias

FromTo: TypeAlias = tuple[str, str]
```

abc.Iterable versus abc.Sequence

Both `math.fsum` and `replacer.zip_replace` must iterate over the entire `Iterable` arguments to return a result. Given an endless iterable such as the `itertools.cycle` generator as input, these functions would consume all memory and crash the Python process. Despite this potential danger, it is fairly common in modern Python to offer functions that accept an `Iterable` input even if they must process it completely to return a result. That gives the caller the option of providing input data as a generator instead of a prebuilt sequence, potentially saving a lot of memory if the number of input items is large.

On the other hand, the `columnize` function from Example 8-13 needs a `Sequence` parameter, and not an `Iterable`, because it must get the `len()` of the input to compute the number of rows up front.

Like `Sequence`, `Iterable` is best used as a parameter type. It's too vague as a return type. A function should be more precise about the concrete type it returns.

Closely related to `Iterable` is the `Iterator` type, used as a return type in Example 8-14. We'll get back to it in Chapter 17, which is about generators and classic iterators.

Parameterized Generics and TypeVar

A parameterized generic is a generic type, written as `list[T]`, where `T` is a type variable that will be bound to a specific type with each usage. This allows a parameter type to be reflected on the result type.

Example 8-16 defines `sample`, a function that takes two arguments: a `Sequence` of elements of type `T`, and an `int`. It returns a `list` of elements of the same type `T`, picked at random from the first argument.

Example 8-16 shows the implementation.

Example 8-16. ***sample.py***

```
from collections.abc import Sequence
from random import shuffle
from typing import TypeVar

T = TypeVar('T')

def sample(population: Sequence[T], size: int) -> list[T]:
    if size < 1:
        raise ValueError('size must be >= 1')
    result = list(population)
    shuffle(result)
    return result[:size]
```

Here are two examples of why I used a type variable in `sample`:

- If called with a tuple of type `tuple[int, ...]`—which is *consistent-with* `Sequence[int]`—then the type parameter is `int`, so the return type is `list[int]`.
- If called with a `str`—which is *consistent-with* `Sequence[str]`—then the type parameter is `str`, so the return type is `list[str]`.

WHY IS TYPEVAR NEEDED?

The authors of PEP 484 wanted to introduce type hints by adding the `typing` module and not changing anything else in the language. With clever metaprogramming they could make the `[]` operator work on classes like `Sequence[T]`. But the name of the `T` variable inside the brackets must be defined somewhere—otherwise the Python interpreter would need deep changes to support generic type notation as special use of `[]`. That's why the `typing.TypeVar` constructor is needed: to introduce the variable name in the current namespace. Languages such as Java, C#, and TypeScript don't require the name of type variable to be declared beforehand, so they have no equivalent of Python's `TypeVar` class.

Another example is the `statistics.mode` function from the standard library, which returns the most common data point from a series.

Here is one usage example from the documentation:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

Without using a `TypeVar`, `mode` could have the signature shown in Example 8-17.

Example 8-17. *mode_float.py*: `mode` that operates on `float` and subtypes[13]

```
from collections import Counter
from collections.abc import Iterable

def mode(data: Iterable[float]) -> float:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

Many uses of `mode` involve `int` or `float` values, but Python has other numerical types, and it is desirable that the return type follows the element type of the given `Iterable`. We can improve that signature using `TypeVar`. Let's start with a simple, but wrong, parameterized signature:

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

T = TypeVar('T')

def mode(data: Iterable[T]) -> T:
```

When it first appears in the signature, the type parameter `T` can be any type. The second time it appears, it will mean the same type as the first.

Therefore, every iterable is *consistent-with* `Iterable[T]`, including iterables of unhashable types that `collections.Counter` cannot handle. We need to restrict the possible types assigned to `T`. We'll see two ways of doing that in the next two sections.

Restricted TypeVar

`TypeVar` accepts extra positional arguments to restrict the type parameter. We can improve the signature of `mode` to accept specific number types, like this:

```
from collections.abc import Iterable
from decimal import Decimal
from fractions import Fraction
from typing import TypeVar

NumberT = TypeVar('NumberT', float, Decimal, Fraction)

def mode(data: Iterable[NumberT]) -> NumberT:
```

That's better than before, and it was the signature for `mode` in the *statistics.pyi* stub file on `typeshed` on May 25, 2020.

However, the `statistics.mode` documentation includes this example:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

In a hurry, we could just add `str` to the `NumberT` definition:

```
NumberT = TypeVar('NumberT', float, Decimal, Fraction, str)
```

That certainly works, but `NumberT` is badly misnamed if it accepts `str`. More importantly, we can't keep listing types forever, as we realize `mode` can deal with them. We can do better with another feature of `TypeVar`, introduced next.

Bounded TypeVar

Looking at the body of `mode` in Example 8-17, we see that the `Counter` class is used for ranking. Counter is based on `dict`, therefore the element type of the `data` iterable must be hashable.

At first, this signature may seem to work:

```
from collections.abc import Iterable, Hashable

def mode(data: Iterable[Hashable]) -> Hashable:
```

Now the problem is that the type of the returned item is `Hashable`: an ABC that implements only the `__hash__` method. So the type checker will not let us do anything with the return value except call `hash()` on it. Not very useful.

The solution is another optional parameter of `TypeVar`: the `bound` keyword parameter. It sets an upper boundary for the acceptable types. In Example 8-18, we have `bound=Hashable`, which means the type parameter may be `Hashable` or any *subtype-of* it.[14]

Example 8-18. *mode_hashable.py*: same as Example 8-17, with a more flexible signature

```
from collections import Counter
from collections.abc import Iterable, Hashable
from typing import TypeVar

HashableT = TypeVar('HashableT', bound=Hashable)

def mode(data: Iterable[HashableT]) -> HashableT:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

To summarize:

- A restricted type variable will be set to one of the types named in the `TypeVar` declaration.
- A bounded type variable will be set to the inferred type of the expression—as long as the inferred type is *consistent-with* the boundary declared in the `bound=` keyword argument of `TypeVar`.

NOTE

It is unfortunate that the keyword argument to declare a bounded `TypeVar` is named `bound=`, because the verb "to bind" is commonly used to mean setting the value of a variable, which in the reference semantics of Python is best described as binding a name to the value. It would have been less confusing if the keyword argument was named `boundary=`.

The `typing.TypeVar` constructor has other optional parameters—`covariant` and `contravariant`—that we'll cover in Chapter 15, "Variance".

Let's conclude this introduction to `TypeVar` with `AnyStr`.

The AnyStr predefined type variable

The `typing` module includes a predefined `TypeVar` named `AnyStr`. It's defined like this:

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

`AnyStr` is used in many functions that accept either `bytes` or `str`, and return values of the given type.

Now, on to `typing.Protocol`, a new feature of Python 3.8 that can support more Pythonic use of type hints.

Static Protocols

NOTE

In object-oriented programming, the concept of a "protocol" as an informal interface is as old as Smalltalk, and is an essential part of Python from the beginning. However, in the context of type hints, a protocol is a `typing.Protocol` subclass defining an interface that a type checker can verify. Both kinds of protocols are covered in Chapter 13. This is just a brief introduction in the context of function annotations.

The `Protocol` type, as presented in PEP 544—Protocols: Structural subtyping (static duck typing), is similar to interfaces in Go: a protocol type is

defined by specifying one or more methods, and the type checker verifies that those methods are implemented where that protocol type is required.

In Python, a protocol definition is written as a `typing.Protocol` subclass. However, classes that *implement* a protocol don't need to inherit, register, or declare any relationship with the class that *defines* the protocol. It's up to the type checker to find the available protocol types and enforce their usage.

Here is a problem that can be solved with the help of `Protocol` and `TypeVar`. Suppose you want to create a function `top(it, n)` that returns the largest `n` elements of the iterable `it`:

```
>>> top([4, 1, 5, 2, 6, 7, 3], 3)
[7, 6, 5]
>>> l = 'mango pear apple kiwi banana'.split()
>>> top(l, 3)
['pear', 'mango', 'kiwi']
>>>
>>> l2 = [(len(s), s) for s in l]
>>> l2
[(5, 'mango'), (4, 'pear'), (5, 'apple'), (4, 'kiwi'), (6, 'banana')]
>>> top(l2, 3)
[(6, 'banana'), (5, 'mango'), (5, 'apple')]
```

A parameterized generic `top` would look like what's shown in Example 8-19.

Example 8-19. `top` function with an undefined `T` type parameter

```
def top(series: Iterable[T], length: int) -> list[T]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]
```

The problem is how to constrain `T`? It cannot be `Any` or `object`, because the `series` must work with `sorted`. The `sorted` built-in actually accepts `Iterable[Any]`, but that's because the optional parameter `key` takes a function that computes an arbitrary sort key from each element. What happens if you give `sorted` a list of plain objects but don't provide a `key` argument? Let's try that:

```
>>> l = [object() for _ in range(4)]
>>> l
[<object object at 0x10fc2fca0>, <object object at 0x10fc2fbb0>,
<object object at 0x10fc2fbc0>, <object object at 0x10fc2fbd0>]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'object' and 'object'
```

The error message shows that `sorted` uses the `<` operator on the elements of the iterable. Is this all it takes? Let's do another quick experiment:[15]

```
>>> class Spam:
...     def __init__(self, n): self.n = n
...     def __lt__(self, other): return self.n < other.n
...     def __repr__(self): return f'Spam({self.n})'
...
>>> l = [Spam(n) for n in range(5, 0, -1)]
>>> l
[Spam(5), Spam(4), Spam(3), Spam(2), Spam(1)]
>>> sorted(l)
[Spam(1), Spam(2), Spam(3), Spam(4), Spam(5)]
```

That confirms it: I can `sort` a list of `Spam` because `Spam` implements `__lt__`—the special method that supports the `<` operator.

So the `T` type parameter in Example 8-19 should be limited to types that implement `__lt__`. In Example 8-18 we needed a type parameter that implemented `__hash__`, so we were able to use `typing.Hashable` as the upper bound for the type parameter. But now there is no suitable type in `typing` or `abc` to use, so we need to create it.

Example 8-20 shows the new `SupportsLessThan` type, a `Protocol`.

Example 8-20. *comparable.py*: definition of a `SupportsLessThan` `Protocol` type

```
from typing import Protocol, Any

class SupportsLessThan(Protocol):  ❶
    def __lt__(self, other: Any) -> bool: ...  ❷
```

❶ A protocol is a subclass of `typing.Protocol`.

❷ The body of the protocol has one or more method definitions, with `...` in their bodies.

A type `T` is *consistent-with* a protocol `P` if `T` implements all the methods defined in `P`, with matching type signatures.

Given `SupportsLessThan`, we can now define this working version of `top` in Example 8-21.

Example 8-21. *top.py*: definition of the `top` function using a `TypeVar` with `bound=SupportsLessThan`

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

from comparable import SupportsLessThan

LT = TypeVar('LT', bound=SupportsLessThan)

def top(series: Iterable[LT], length: int) -> list[LT]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]
```

Let's test-drive `top`. Example 8-22 shows part of a test suite for use with `pytest`. It tries calling `top` first with a generator expression that yields `tuple[int, str]`, and then with a list of `object`. With the list of `object`, we expect to get a `TypeError` exception.

Example 8-22. *top_test.py*: partial listing of the test suite for `top`

```
from collections.abc import Iterator
from typing import TYPE_CHECKING  ❶

import pytest

from top import top

# several lines omitted

def test_top_tuples() -> None:
    fruit = 'mango pear apple kiwi banana'.split()
    series: Iterator[tuple[int, str]] = (  ❷
```

```
            (len(s), s) for s in fruit)
    length = 3
    expected = [(6, 'banana'), (5, 'mango'), (5, 'apple')]
    result = top(series, length)
    if TYPE_CHECKING:  ❸
        reveal_type(series)  ❹
        reveal_type(expected)
        reveal_type(result)
    assert result == expected

# intentional type error
def test_top_objects_error() -> None:
    series = [object() for _ in range(4)]
    if TYPE_CHECKING:
        reveal_type(series)
    with pytest.raises(TypeError) as excinfo:
        top(series, 3)  ❺
    assert "'<' not supported" in str(excinfo.value)
```

❶ The `typing.TYPE_CHECKING` constant is always `False` at runtime, but type checkers pretend it is `True` when they are type checking.

❷ Explicit type declaration for the `series` variable, to make the Mypy output easier to read.[16]

❸ This `if` prevents the next three lines from executing when the test runs.

❹ `reveal_type()` cannot be called at runtime, because it is not a regular function but a Mypy debugging facility—that's why there is no `import` for it. Mypy will output one debugging message for each `reveal_type()` pseudofunction call, showing the inferred type of the argument.

❺ This line will be flagged as an error by Mypy.

The preceding tests pass—but they would pass anyway, with or without type hints in *top.py*. More to the point, if I check that test file with Mypy, I see that the `TypeVar` is working as intended. See the `mypy` command output in Example 8-23.

WARNING

As of Mypy 0.910 (July 2021), the output of `reveal_type` does not show precisely the types I declared in some cases, but compatible types instead. For example, I did not use `typing.Iterator` but used `abc.Iterator`. Please ignore this detail. The Mypy output is still useful. I will pretend this issue of Mypy is fixed when discussing the output.

Example 8-23. Output of *mypy top_test.py* (lines split for readability)

```
…/comparable/ $ mypy top_test.py
top_test.py:32: note:
    Revealed type is "typing.Iterator[Tuple[builtins.int, builtins.str]]" ❶
top_test.py:33: note:
    Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]"
top_test.py:34: note:
    Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]" ❷
top_test.py:41: note:
    Revealed type is "builtins.list[builtins.object*]" ❸
top_test.py:43: error:
    Value of type variable "LT" of "top" cannot be "object"  ❹
Found 1 error in 1 file (checked 1 source file)
```

❶ In `test_top_tuples`, `reveal_type(series)` shows it is an `Iterator[tuple[int, str]]`—which I explicitly declared.

❷ `reveal_type(result)` confirms that the type returned by the `top` call is what I wanted: given the type of `series`, the `result` is `list[tuple[int, str]]`.

❸ In `test_top_objects_error`, `reveal_type(series)` shows it is `list[object*]`. Mypy puts a `*` after any type that was inferred: I did not annotate the type of `series` in this test.

❹ Mypy flags the error that this test intentionally triggers: the element type of the `Iterable series` cannot be `object` (it must be of type `SupportsLessThan`).

A key advantage of a protocol type over ABCs is that a type doesn't need any special declaration to be *consistent-with* a protocol type. This allows a protocol to be created leveraging preexisting types, or types implemented in code that we do not control. I don't need to derive or register `str`,

`tuple`, `float`, `set`, etc. with `SupportsLessThan` to use them where a `SupportsLessThan` parameter is expected. They only need to implement `__lt__`. And the type checker will still be able do its job, because `SupportsLessThan` is explicitly defined as a `Protocol`—in contrast with the implicit protocols that are common with duck typing, which are invisible to the type checker.

The special `Protocol` class was introduced in PEP 544—Protocols: Structural subtyping (static duck typing). Example 8-21 demonstrates why this feature is known as *static duck typing*: the solution to annotate the `series` parameter of `top` was to say "The nominal type of `series` doesn't matter, as long as it implements the `__lt__` method." Python's duck typing always allowed us to say that implicitly, leaving static type checkers clueless. A type checker can't read CPython's source code in C, or perform console experiments to find out that `sorted` only requires that the elements support `<`.

Now we can make duck typing explicit for static type checkers. That's why it makes sense to say that `typing.Protocol` gives us *static duck typing*.[17]

There's more to see about `typing.Protocol`. We'll come back to it in Part IV, where Chapter 13 contrasts structural typing, duck typing, and ABCs—another approach to formalizing protocols. In addition, "Overloaded Signatures" (Chapter 15) explains how to declare overloaded function signatures with `@typing.overload`, and includes an extensive example using `typing.Protocol` and a bounded `TypeVar`.

NOTE

`typing.Protocol` makes it possible to annotate the `double` function presented in "Types Are Defined by Supported Operations" without losing functionality. The key is to define a protocol class with the `__mul__` method. I invite you to do that as an exercise. The solution appears in "The Typed double Function" (Chapter 13).

Callable

To annotate callback parameters or callable objects returned by higher-order functions, the `collections.abc` module provides the `Callable`

type, available in the `typing` module for those not yet using Python 3.9. A `Callable` type is parameterized like this:

```
Callable[[ParamType1, ParamType2], ReturnType]
```

The parameter list— `[ParamType1, ParamType2]` —can have zero or more types.

Here is an example in the context of a `repl` function, part of a simple interactive interpreter we'll see in "Pattern Matching in lis.py: A Case Study":[18]

```
def repl(input_fn: Callable[[Any], str] = input]) -> None:
```

During normal usage, the `repl` function uses Python's `input` built-in to read expressions from the user. However, for automated testing or for integration with other input sources, `repl` accepts an optional `input_fn` parameter: a `Callable` with the same parameter and return types as `input`.

The built-in `input` has this signature on typeshed:

```
def input(__prompt: Any = ...) -> str: ...
```

The `input` signature is *consistent-with* this `Callable` type hint:

```
Callable[[Any], str]
```

There is no syntax to annotate optional or keyword argument types. The documentation of `typing.Callable` says "such function types are rarely used as callback types." If you need a type hint to match a function with a flexible signature, replace the whole parameter list with `...` — like this:

```
Callable[..., ReturnType]
```

The interaction of generic type parameters with a type hierarchy introduces a new typing concept: variance.

Variance in Callable types

Imagine a temperature control system with a simple `update` function as shown in Example 8-24. The `update` function calls the `probe` function to get the current temperature, and calls `display` to show the temperature to the user. Both `probe` and `display` are passed as arguments to `update` for didactic reasons. The goal of the example is to contrast two `Callable` annotations: one with a return type, the other with a parameter type.

Example 8-24. Illustrating variance.

```
from collections.abc import Callable

def update(  ❶
        probe: Callable[[], float],  ❷
        display: Callable[[float], None]  ❸
    ) -> None:
    temperature = probe()
    # imagine lots of control code here
    display(temperature)

def probe_ok() -> int:  ❹
    return 42

def display_wrong(temperature: int) -> None:  ❺
    print(hex(temperature))

update(probe_ok, display_wrong)  # type error  ❻

def display_ok(temperature: complex) -> None:  ❼
    print(temperature)

update(probe_ok, display_ok)  # OK  ❽
```

❶ `update` takes two callables as arguments.

❷ `probe` must be a callable that takes no arguments and returns a `float`.

❸ `display` takes a `float` argument and returns `None`.

❹

`probe_ok` is *consistent-with* `Callable[[], float]` because returning an `int` does not break code that expects a `float`.

❺ `display_wrong` is not *consistent-with* `Callable[[float], None]` because there's no guarantee that a function that expects an `int` can handle a `float`; for example, Python's `hex` function accepts an `int` but rejects a `float`.

❻ Mypy flags this line because `display_wrong` is incompatible with the type hint in the `display` parameter of `update`.

❼ `display_ok` is *consistent-with* `Callable[[float], None]` because a function that accepts a `complex` can also handle a `float` argument.

❽ Mypy is happy with this line.

To summarize, it's OK to provide a callback that returns an `int` when the code expects a callback that returns a `float`, because an `int` value can always be used where a `float` is expected.

Formally, we say that `Callable[[], int]` is *subtype-of* `Callable[[], float]`—as `int` is *subtype-of* `float`. This means that `Callable` is *covariant* on the return type because the *subtype-of* relationship of the types `int` and `float` is in the same direction as the relationship of the `Callable` types that use them as return types.

On the other hand, it's a type error to provide a callback that takes a `int` argument when a callback that handles a `float` is required.

Formally, `Callable[[int], None]` is not a *subtype-of* `Callable[[float], None]`. Although `int` is *subtype-of* `float`, in the parameterized `Callable` type the relationship is reversed: `Callable[[float], None]` is *subtype-of* `Callable[[int], None]`. Therefore we say that `Callable` is *contravariant* on the declared parameter types.

"Variance" in Chapter 15 explains variance with more details and examples of invariant, covariant, and contravariant types.

TIP

For now, rest assured that most parameterized generic types are *invariant*, therefore simpler. For example, if I declare `scores: list[float]`, that tells me exactly what I can assign to `scores`. I can't assign objects declared as `list[int]` or `list[complex]`:

- A `list[int]` object is not acceptable because it cannot hold `float` values which my code may need to put into `scores`.
- A `list[complex]` object is not acceptable because my code may need to sort `scores` to find the median, but `complex` does not provide `__lt__`, therefore `list[complex]` is not sortable.

Now we get to the last special type we'll cover in this chapter.

NoReturn

This is a special type used only to annotate the return type of functions that never return. Usually, they exist to raise exceptions. There are dozens of such functions in the standard library.

For example, `sys.exit()` raises `SystemExit` to terminate the Python process.

Its signature in `typeshed` is:

```
def exit(__status: object = ...) -> NoReturn: ...
```

The `__status` parameter is positional only, and it has a default value. Stub files don't spell out the default values, they use `...` instead. The type of `__status` is `object`, which means it may also be `None`, therefore it would be redundant to mark it `Optional[object]`.

In Chapter 24, Example 24-6 uses `NoReturn` in the `__flag_unknown_attrs`, a method designed to produce a user-friendly and comprehensive error message, and then raise `AttributeError`.

The last section in this epic chapter is about positional and variadic parameters.

Annotating Positional Only and Variadic Parameters

Recall the `tag` function from Example 7-9. The last time we saw its signature was in "Positional-Only Parameters":

```
def tag(name, /, *content, class_=None, **attrs):
```

Here is `tag`, fully annotated, written in several lines—a common convention for long signatures, with line breaks the way the *blue* formatter would do it:

```
from typing import Optional

def tag(
    name: str,
    /,
    *content: str,
    class_: Optional[str] = None,
    **attrs: str,
) -> str:
```

Note the type hint `*content: str` for the arbitrary positional parameters; this means all those arguments must be of type `str`. The type of the `content` local variable in the function body will be `tuple[str, ...]`.

The type hint for the arbitrary keyword arguments is `**attrs: str` in this example, therefore the type of `attrs` inside the function will be `dict[str, str]`. For a type hint like `**attrs: float`, the type of `attrs` in the function would be `dict[str, float]`.``

If the `attrs` parameter must accept values of different types, you'll need to use a `Union[]` or `Any`: `**attrs: Any`.

The `/` notation for positional-only parameters is only available in Python ≥ 3.8. In Python 3.7 or earlier, that's a syntax error. The PEP 484 convention is to prefix each positional-only parameter name with two underscores. Here is the `tag` signature again, now in two lines, using the PEP 484 convention:

```
from typing import Optional

def tag(__name: str, *content: str, class_: Optional[str] = None,
        **attrs: str) -> str:
```

Mypy understands and enforces both ways of declaring positional-only parameters.

To close this chapter, let's briefly consider the limits of type hints and the static type system they support.

Imperfect Typing and Strong Testing

Maintainers of large corporate codebases report that many bugs are found by static type checkers and fixed more cheaply than if the bugs were discovered only after the code is running in production. However, it's essential to note that automated testing was standard practice and widely adopted long before static typing was introduced in the companies that I know about.

Even in the contexts where they are most beneficial, static typing cannot be trusted as the ultimate arbiter of correctness. It's not hard to find:

False positives

Tools report type errors on code that is correct.

False negatives

Tools don't report type errors on code that is incorrect.

Also, if we are forced to type check everything, we lose some of the expressive power of Python:

- Some handy features can't be statically checked; for example, argument unpacking like `config(**settings)`.
- Advanced features like properties, descriptors, metaclasses, and metaprogramming in general are poorly supported or beyond comprehension for type checkers.
- Type checkers lag behind Python releases, rejecting or even crashing while analyzing code with new language features—for more than a year in some cases.

Common data constraints cannot be expressed in the type system—even simple ones. For example, type hints are unable to ensure "quantity must be an integer > 0" or "label must be a string with 6 to 12 ASCII letters." In general, type hints are not helpful to catch errors in business logic.

Given those caveats, type hints cannot be the mainstay of software quality, and making them mandatory without exception would amplify the downsides.

Consider a static type checker as one of the tools in a modern CI pipeline, along with test runners, linters, etc. The point of a CI pipeline is to reduce software failures, and automated tests catch many bugs that are beyond the reach of type hints. Any code you can write in Python, you can test in Python—with or without type hints.

NOTE

The title and conclusion of this section were inspired by Bruce Eckel's article "Strong Typing vs. Strong Testing", also published in the anthology *The Best Software Writing I*, edited by Joel Spolsky (Apress). Bruce is a fan of Python and author of books about C++, Java, Scala, and Kotlin. In that post, he tells how he was a static typing advocate until he learned Python and concluded: "If a Python program has adequate unit tests, it can be as robust as a C++, Java, or C# program with adequate unit tests (although the tests in Python will be faster to write)."

This wraps up our coverage of Python's type hints for now. They are also the main focus of Chapter 15, which covers generic classes, variance, overloaded signatures, type casting, and more. Meanwhile, type hints will make guest appearances in several examples throughout the book.

Chapter Summary

We started with a brief introduction to the concept of gradual typing and then switched to a hands-on approach. It's hard to see how gradual typing works without a tool that actually reads the type hints, so we developed an annotated function guided by Mypy error reports.

Back to the idea of gradual typing, we explored how it is a hybrid of Python's traditional duck typing and the nominal typing more familiar to users of Java, C++, and other statically typed languages.

Most of the chapter was devoted to presenting the major groups of types used in annotations. Many of the types we covered are related to familiar Python object types, such as collections, tuples, and callables—extended to support generic notation like `Sequence[float]`. Many of those types are temporary surrogates implemented in the `typing` module before the standard types were changed to support generics in Python 3.9.

Some of the types are special entities. `Any`, `Optional`, `Union`, and `NoReturn` have nothing to do with actual objects in memory, but exist only in the abstract domain of the type system.

We studied parameterized generics and type variables, which bring more flexibility to type hints without sacrificing type safety.

Parameterized generics become even more expressive with the use of `Protocol`. Because it appeared only in Python 3.8, `Protocol` is not widely used yet—but it is hugely important. `Protocol` enables static duck typing: the essential bridge between Python's duck-typed core and the nominal typing that allows static type checkers to catch bugs.

While covering some of these types, we experimented with Mypy to see type checking errors and inferred types with the help of Mypy's magic `reveal_type()` function.

The final section covered how to annotate positional-only and variadic parameters.

Type hints are a complex and evolving topic. Fortunately, they are an optional feature. Let us keep Python accessible to the widest user base and stop preaching that all Python code should have type hints—as I've seen in public sermons by typing evangelists.

Our BDFL[19] emeritus led this push toward type hints in Python, so it's only fair that this chapter starts and ends with his words:

> *I wouldn't like a version of Python where I was morally obligated to add type hints all the time. I really do think that type hints have their place but there are also plenty of times that it's not worth it, and it's so wonderful that you can choose to use them.*[20]
>
> —Guido van Rossum

Further Reading

Bernát Gábor wrote in his excellent post, "The state of type hints in Python":

> *Type hints should be used whenever unit tests are worth writing.*

I am a big fan of testing, but I also do a lot of exploratory coding. When I am exploring, tests and type hints are not helpful. They are a drag.

Gábor's post is one of the best introductions to Python's type hints that I found, along with Geir Arne Hjelle's "Python Type Checking (Guide)". "Hypermodern Python Chapter 4: Typing" by Claudio Jolowicz is a shorter introduction that also covers runtime type checking validation.

For deeper coverage, the Mypy documentation is the best source. It is valuable regardless of the type checker you are using, because it has tutorial and reference pages about Python typing in general—not just about the Mypy tool itself. There you will also find a handy cheat sheets and a very useful page about common issues and solutions.

The `typing` module documentation is a good quick reference, but it doesn't go into much detail. PEP 483—The Theory of Type Hints includes a deep explanation about variance, using `Callable` to illustrate contravariance. The ultimate references are the PEP documents related to typing. There are more than 20 of them already. The intended audience of PEPs are Python core developers and Python's Steering Council, so they assume a lot of prior knowledge and are certainly not light reading.

As mentioned, Chapter 15 covers more typing topics, and "Further Reading" provides additional references, including Table 15-1, listing typing PEPs approved or under discussion as of late 2021.

"Awesome Python Typing" is a valuable collection of links to tools and references.

SOAPBOX

Just Ride

> *Forget the ultralight, uncomfortable bikes, flashy jerseys, clunky shoes that clip onto tiny pedals, the grinding out of endless miles. Instead, ride like you did when you were a kid—just get on your bike and discover the pure joy of riding it.*
>
> —Grant Petersen, *Just Ride: A Radically Practical Guide to Riding Your Bike* (Workman Publishing)

If coding is not your whole profession, but a useful tool in your profession, or something you do to learn, tinker, and enjoy, you probably don't need type hints any more than most bikers need shoes with stiff soles and metal cleats.

Just code.

The Cognitive Effect of Typing

I worry about the effect type hints will have on Python coding style.

I agree that users of most APIs benefit from type hints. But Python attracted me—among other reasons—because it provides functions that are so powerful that they replace entire APIs, and we can write similarly powerful functions ourselves. Consider the `max()` built-in. It's powerful, yet easy to understand. But I will show in "Max Overload" that it takes 14 lines of type hints to properly annotate it—not counting a `typing.Protocol` and a few `TypeVar` definitions to support those type hints.

I am concerned that strict enforcement of type hints in libraries will discourage programmers from even considering writing such functions in the future.

According to the English Wikipedia, "linguistic relativity"—a.k.a. the Sapir–Whorf hypothesis— is a "principle claiming that the structure of a language affects its speakers' world view or cognition." Wikipedia further explains:

- The *strong* version says that language *determines* thought and that linguistic categories limit and determine cognitive categories.
- The *weak* version says that linguistic categories and usage only *influence* thought and decisions.

Linguists generally agree the strong version is false, but there is empirical evidence supporting the weak version.

I am not aware of specific studies with programming languages, but in my experience they've had a big impact on how I approach problems. The first programming language I used professionally was Applesoft BASIC in the age of 8-bit computers. Recursion was not directly supported by BASIC—you had to roll your own call stack to use it. So I never considered using recursive algorithms or data structures. I knew at some conceptual level such things existed, but they weren't part of my problem-solving toolbox.

Decades later when I started with Elixir, I enjoyed solving problems with recursion and overused it—until I discovered that many of my solutions would be simpler if I used existing functions from the Elixir `Enum` and `Stream` modules. I learned that idiomatic Elixir application-level code rarely has explicit recursive calls, but uses enums and streams that implement recursion under the hood.

Linguistic relativity could explain the widespread idea (also unproven) that learning different programming languages makes you a better programmer, particularly when the languages support different programming paradigms. Practicing Elixir made me more likely to apply functional patterns when I write Python or Go code.

Now, back to Earth.

The `requests` package would probably have a very different API if Kenneth Reitz was determined (or told by his boss) to annotate all its functions. His goal was to write an API that was easy to use, flexible, and powerful. He succeeded, given the amazing popularity of `requests`—in May 2020, it's #4 on PyPI Stats, with 2.6 million downloads a day. #1 is `urllib3`, a dependency of `requests`.

In 2017, the `requests` maintainers decided not to spend their time writing type hints. One of the maintainers, Cory Benfield, had written an e-mail stating:

> *I think that libraries with Pythonic APIs are the least likely to take up this typing system because it will provide the least value to them.*

In that message, Benfield gave this extreme example of a tentative type definition for the `files` keyword argument of `requests.request()`:

```
Optional[
  Union[
    Mapping[
      basestring,
      Union[
        Tuple[basestring, Optional[Union[basestring, file]]],
        Tuple[basestring, Optional[Union[basestring, file]],
              Optional[basestring]],
        Tuple[basestring, Optional[Union[basestring, file]],
              Optional[basestring], Optional[Headers]]
      ]
    ],
    Iterable[
      Tuple[
        basestring,
        Union[
          Tuple[basestring, Optional[Union[basestring, file]]],
          Tuple[basestring, Optional[Union[basestring, file]],
                Optional[basestring]],
          Tuple[basestring, Optional[Union[basestring, file]],
                Optional[basestring], Optional[Headers]]
      ]
    ]
  ]
]
```

And that assumes this definition:

```
Headers = Union[
  Mapping[basestring, basestring],
  Iterable[Tuple[basestring, basestring]],
]
```

Do you think `requests` would be the way it is if the maintainers insisted on 100% type hint coverage? SQLAlchemy is another important package that doesn't play well with type hints.

What makes these libraries great is embracing the dynamic nature of Python.

While there are benefits to type hints, there is also a price to pay.

First, there is the significant investment of understanding how the type system works. That's a one-time cost.

But there is also a recurring cost, forever.

We lose some of the expressive power of Python if we insist on type checking everything. Beautiful features like argument unpacking—e.g., `config(**settings)` —are beyond comprehension for type checkers.

If you want to have a call like `config(**settings)` type checked, you must spell every argument out. That brings me memories of Turbo Pascal code I wrote 35 years ago.

Libraries that use metaprogramming are hard or impossible to annotate. Surely metaprogramming can be abused, but it's also what makes many Python packages a joy to use.

If type hints are mandated top-down without exceptions in large companies, I bet soon we'll see people using code generation to reduce boilerplate in Python source-code—a common practice with less dynamic languages.

For some projects and contexts, type hints just don't make sense. Even in contexts where they mostly make sense, they don't make sense all the time. Any reasonable policy about the use of type hints must have exceptions.

Alan Kay, the Turing Award laureate who pioneered object-oriented programming, once said:

> *Some people are completely religious about type systems and as a mathematician I love the idea of type systems, but nobody has ever come up with one that has enough scope.*[21]

Thank Guido for optional typing. Let's use it as intended, and not aim to annotate everything into strict conformity to a coding style that looks like Java 1.5.

Duck Typing FTW

Duck typing fits my brain, and static duck typing is a good compromise allowing static type checking without losing a lot of flexibility that some nominal type systems only provide with a lot of complexity—if ever.

Before PEP 544, this whole idea of type hints seemed utterly unPythonic to me. I was very glad to see `typing.Protocol` land in Python. It brings balance to the force.

Generics or Specifics?

From a Python perspective, the typing usage of the term "generic" is backward. Common meanings of "generic" are "applicable to an entire class or group" or "without a brand name."

Consider `list` versus `list[str]`. The first is generic: it accepts any object. The second is specific: it only accepts `str`.

The term makes sense in Java, though. Before Java 1.5, all Java collections (except the magic `array`) were "specific": they could only hold `Object` references, so we had to cast the items that came out of a collection to use them. With Java 1.5, collections got type parameters, and became "generic."

---

1 PEP 484—Type Hints, "Rationale and Goals"; bold emphasis retained from the original.

2 A just-in-time compiler like the one in PyPy has much better data than type hints: it monitors the Python program as it runs, detects the concrete types in use, and generates optimized machine code for those concrete types.

3 For example, recursive types are not supported as of July 2021—see `typing` module issue #182, Define a JSON type and Mypy issue #731, Support recursive types.

4 Python doesn't provide syntax to control the set of possible values for a type—except in `Enum` types. For example, using type hints you can't define `Quantity` as an integer between 1 and 1000, or `AirportCode` as a 3-letter combination. NumPy offers `uint8`, `int16`, and other machine-oriented numeric types, but in the Python standard library we only have types with very small sets of values (`NoneType`, `bool`) or extremely large sets (`float`, `int`, `str`, all possible tuples, etc.).

5 Duck typing is an implicit form of *structural typing*, which Python ≥ 3.8 also supports with the introduction of `typing.Protocol`. This is covered later in this chapter—in "Static Protocols"—with more details in Chapter 13.

6 Inheritance is often overused and hard to justify in examples that are realistic yet simple, so please accept this animal example as a quick illustration of subtyping.

7 MIT Professor, programming language designer, and Turing Award recipient. Wikipedia: Barbara Liskov.

8 To be more precise, `ord` only accepts `str` or `bytes` with `len(s) == 1`. But the type system currently can't express this constraint.

9 In ABC—the language that most influenced the initial design of Python—each list was constrained to accept values of a single type: the type of the first item you put into it.

10 One of my contributions to the `typing` module documentation was to add dozens of deprecation warnings as I reorganized the entries below "Module Contents" into subsections, under the supervision of Guido van Rossum.

11 I use `:=` when it makes sense in a few examples, but I don't cover it in the book. Please see PEP 572—Assignment Expressions for all the gory details.

12 Actually, `dict` is a virtual subclass of `abc.MutableMapping`. The concept of a virtual subclass is explained in Chapter 13. For now, know that `issubclass(dict, abc.MutableMapping)` is `True`, despite the fact that `dict` is implemented in C and does not inherit anything from `abc.MutableMapping`, but only from `object`.

13 The implementation here is simpler than the one in the Python standard library `statistics` module.

14 I contributed this solution to `typeshed`, and that's how `mode` is annotated on *statistics.pyi* as of May 26, 2020.

15 How wonderful it is to open an interactive console and rely on duck typing to explore language features like I just did. I badly miss this kind of exploration when I use languages that don't support it.

16 Without this type hint, Mypy would infer the type of `series` as `Generator[Tuple[builtins.int, builtins.str*], None, None]`, which is verbose but *consistent-with* `Iterator[tuple[int, str]]`, as we'll see in "Generic Iterable Types".

17 I don't know who invented the term *static duck typing*, but it became more popular with the Go language, which has interface semantics that are more like Python's protocols than the nominal interfaces of Java.

18 REPL stands for Read-Eval-Print-Loop, the basic behavior of interactive interpreters.

19 “Benevolent Dictator For Life.” See Guido van van Rossum on the “Origin of BDFL”.

20 From the YouTube video, “Type Hints by Guido van Rossum (March 2015)”. Quote starts at 13’40”. I did some light editing for clarity.

21 Source: “A Conversation with Alan Kay”.