CHAPTER 15

More About Type Hints

> I learned a painful lesson that for small programs, dynamic typing is great. For large programs you need a more disciplined approach. And it helps if the language gives you that discipline rather than telling you "Well, you can do whatever you want".
>
> —Guido van Rossum, a fan of Monty Python[1]

This chapter is a sequel to Chapter 8, covering more of Python's gradual type system. The main topics are:

- Overloaded function signatures
- `typing.TypedDict` for type hinting `dicts` used as records
- Type casting
- Runtime access to type hints
- Generic types
  - — Declaring a generic class
  - — Variance: invariant, covariant, and contravariant types
  - — Generic static protocols

What's New in This Chapter

This chapter is new in the second edition of *Fluent Python*. Let's start with overloads.

1 From YouTube video of "A Language Creators' Conversation: Guido van Rossum, James Gosling, Larry Wall, and Anders Hejlsberg," streamed live on April 2, 2019. Quote starts at 1:32:05, edited for brevity. Full transcript available at *https://github.com/fluentpython/language-creators*.

Overloaded Signatures

Python functions may accept different combinations of arguments. The `@typing.overload` decorator allows annotating those different combinations. This is particularly important when the return type of the function depends on the type of two or more parameters.

Consider the `sum` built-in function. This is the text of `help(sum)`:

```
>>> help(sum)
sum(iterable, /, start=0)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

The `sum` built-in is written in C, but *typeshed* has overloaded type hints for it, in *builtins.pyi*:

```
@overload
def sum(__iterable: Iterable[_T]) -> Union[_T, int]: ...
@overload
def sum(__iterable: Iterable[_T], start: _S) -> Union[_T, _S]: ...
```

First let's look at the overall syntax of overloads. That's all the code about the `sum` you'll find in the stub file (*.pyi*). The implementation would be in a different file. The ellipsis (`...`) has no function other than to fulfill the syntactic requirement for a function body, similar to `pass`. So *.pyi* files are valid Python files.

As mentioned in "Annotating Positional Only and Variadic Parameters" on page 295, the two leading underscores in `__iterable` are a PEP 484 convention for positional-only arguments that is enforced by Mypy. It means you can call `sum(my_list)`, but not `sum(__iterable = my_list)`.

The type checker tries to match the given arguments with each overloaded signature, in order. The call `sum(range(100), 1000)` doesn't match the first overload, because that signature has only one parameter. But it matches the second.

You can also use `@overload` in a regular Python module, by writing the overloaded signatures right before the function's actual signature and implementation. Example 15-1 shows how `sum` would appear annotated and implemented in a Python module.

Example 15-1. mysum.py: definition of the `sum` function with overloaded signatures

```
import functools
import operator
from collections.abc import Iterable
```

```
from typing import overload, Union, TypeVar

T = TypeVar('T')
S = TypeVar('S')  ❶

@overload
def sum(it: Iterable[T]) -> Union[T, int]: ...  ❷
@overload
def sum(it: Iterable[T], /, start: S) -> Union[T, S]: ...  ❸
def sum(it, /, start=0):  ❹
    return functools.reduce(operator.add, it, start)
```

❶ We need this second `TypeVar` in the second overload.

❷ This signature is for the simple case: `sum(my_iterable)`. The result type may be `T`—the type of the elements that `my_iterable` yields—or it may be `int` if the iterable is empty, because the default value of the `start` parameter is `0`.

❸ When `start` is given, it can be of any type `S`, so the result type is `Union[T, S]`. This is why we need `S`. If we reused `T`, then the type of `start` would have to be the same type as the elements of `Iterable[T]`.

❹ The signature of the actual function implementation has no type hints.

That's a lot of lines to annotate a one-line function. Probably overkill, I know. At least it wasn't a `foo` function.

If you want to learn about `@overload` by reading code, *typeshed* has hundreds of examples. On *typeshed*, the stub file for Python's built-ins has 186 overloads as I write this—more than any other in the standard library.

Take Advantage of Gradual Typing

Aiming for 100% of annotated code may lead to type hints that add lots of noise but little value. Refactoring to simplify type hinting can lead to cumbersome APIs. Sometimes it's better to be pragmatic and leave a piece of code without type hints.

The handy APIs we call Pythonic are often hard to annotate. In the next section we'll see an example: six overloads are needed to properly annotate the flexible `max` built-in function.

Max Overload

It is difficult to add type hints to functions that leverage the powerful dynamic features of Python.

While studying typeshed, I found bug report #4051: Mypy failed to warn that it is illegal to pass `None` as one of the arguments to the built-in `max()` function, or to pass an iterable that at some point yields `None`. In either case, you get a runtime exception like this one:

```
TypeError: '>' not supported between instances of 'int' and 'NoneType'
```

The documentation of `max` starts with this sentence:

> Return the largest item in an iterable or the largest of two or more arguments.

To me, that's a very intuitive description.

But if I must annotate a function described in those terms, I have to ask: which is it? An iterable or two or more arguments?

The reality is more complicated because `max` also takes two optional keyword arguments: `key` and `default`.

I coded `max` in Python to make it easier to see the relationship between how it works and the overloaded annotations (the built-in `max` is in C); see Example 15-2.

Example 15-2. mymax.py: Python rewrite of `max` function

```
# imports and definitions omitted, see next listing

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

# overloaded type hints omitted, see next listing

def max(first, *args, key=None, default=MISSING):
    if args:
        series = args
        candidate = first
    else:
        series = iter(first)
        try:
            candidate = next(series)
        except StopIteration:
            if default is not MISSING:
                return default
            raise ValueError(EMPTY_MSG) from None
    if key is None:
        for current in series:
            if candidate < current:
                candidate = current
    else:
        candidate_key = key(candidate)
        for current in series:
            current_key = key(current)
```

```
            if candidate_key < current_key:
                candidate = current
                candidate_key = current_key
    return candidate
```

The focus of this example is not the logic of `max`, so I will not spend time with its implementation, other than explaining `MISSING`. The `MISSING` constant is a unique `object` instance used as a sentinel. It is the default value for the `default=` keyword argument, so that `max` can accept `default=None` and still distinguish between these two situations:

1. The user did not provide a value for `default=`, so it is `MISSING`, and `max` raises `ValueError` if `first` is an empty iterable.
2. The user provided some value for `default=`, including `None`, so `max` returns that value if `first` is an empty iterable.

To fix issue #4051, I wrote the code in Example 15-3.[2]

Example 15-3. mymax.py: top of the module, with imports, definitions, and overloads

```
from collections.abc import Callable, Iterable
from typing import Protocol, Any, TypeVar, overload, Union

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

T = TypeVar('T')
LT = TypeVar('LT', bound=SupportsLessThan)
DT = TypeVar('DT')

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

@overload
def max(__arg1: LT, __arg2: LT, *args: LT, key: None = ...) -> LT:
    ...
@overload
def max(__arg1: T, __arg2: T, *args: T, key: Callable[[T], LT]) -> T:
    ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
```

2 I am grateful to Jelle Zijlstra—a *typeshed* maintainer—who taught me several things, including how to reduce my original nine overloads to six.

```
    ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

My Python implementation of `max` is about the same length as all those typing imports and declarations. Thanks to duck typing, my code has no `isinstance` checks, and provides the same error checking as those type hints—but only at runtime, of course.

A key benefit of `@overload` is declaring the return type as precisely as possible, according to the types of the arguments given. We'll see that benefit next by studying the overloads for `max` in groups of one or two at a time.

Arguments implementing SupportsLessThan, but key and default not provided

```
@overload
def max(__arg1: LT, __arg2: LT, *_args: LT, key: None = ...) -> LT:
    ...
# ... lines omitted ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...
```

In these cases, the inputs are either separate arguments of type `LT` implementing `SupportsLessThan`, or an `Iterable` of such items. The return type of `max` is the same as the actual arguments or items, as we saw in "Bounded TypeVar" on page 284.

Sample calls that match these overloads:

```
max(1, 2, -3)  # returns 2
max(['Go', 'Python', 'Rust'])  # returns 'Rust'
```

Argument key provided, but no default

```
@overload
def max(__arg1: T, __arg2: T, *_args: T, key: Callable[[T], LT]) -> T:
    ...
# ... lines omitted ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
    ...
```

The inputs can be separate items of any type `T` or a single `Iterable[T]`, and `key=` must be a callable that takes an argument of the same type `T`, and returns a value that

implements `SupportsLessThan`. The return type of `max` is the same as the actual arguments.

Sample calls that match these overloads:

```
max(1, 2, -3, key=abs)  # returns -3
max(['Go', 'Python', 'Rust'], key=len)  # returns 'Python'
```

Argument default provided, but no key

```
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
```

The input is an iterable of items of type `LT` implementing `SupportsLessThan`. The `default=` argument is the return value when the `Iterable` is empty. Therefore the return type of `max` must be a `Union` of type `LT` and the type of the `default` argument.

Sample calls that match these overloads:

```
max([1, 2, -3], default=0)  # returns 2
max([], default=None)  # returns None
```

Arguments key and default provided

```
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

The inputs are:

- An `Iterable` of items of any type `T`
- Callable that takes an argument of type `T` and returns a value of type `LT` that implements `SupportsLessThan`
- A default value of any type `DT`

The return type of `max` must be a `Union` of type `T` or the type of the `default` argument:

```
max([1, 2, -3], key=abs, default=None)  # returns -3
max([], key=abs, default=None)  # returns None
```

Takeaways from Overloading max

Type hints allow Mypy to flag a call like `max([None, None])` with this error message:

```
mymax_demo.py:109: error: Value of type variable "_LT" of "max"
  cannot be "None"
```

On the other hand, having to write so many lines to support the type checker may discourage people from writing convenient and flexible functions like `max`. If I had to reinvent the `min` function as well, I could refactor and reuse most of the implementation of `max`. But I'd have to copy and paste all overloaded declarations—even though they would be identical for `min`, except for the function name.

My friend João S. O. Bueno—one of the smartest Python devs I know—tweeted this:

> Although it is this hard to express the signature of `max`—it fits in one's mind quite easily. My understanding is that the expressiveness of annotation markings is very limited, compared to that of Python.

Now let's study the `TypedDict` typing construct. It is not as useful as I imagined at first, but has its uses. Experimenting with `TypedDict` demonstrates the limitations of static typing for handling dynamic structures, such as JSON data.

TypedDict

It's tempting to use `TypedDict` to protect against errors while handling dynamic data structures like JSON API responses. But the examples here make clear that correct handling of JSON must be done at runtime, and not with static type checking. For runtime checking of JSON-like structures using type hints, check out the *pydantic* package on PyPI.

Python dictionaries are sometimes used as records, with the keys used as field names and field values of different types.

For example, consider a record describing a book in JSON or Python:

```
{"isbn": "0134757599",
 "title": "Refactoring, 2e",
 "authors": ["Martin Fowler", "Kent Beck"],
 "pagecount": 478}
```

Before Python 3.8, there was no good way to annotate a record like that, because the mapping types we saw in "Generic Mappings" on page 276 limit all values to have the same type.

Here are two lame attempts to annotate a record like the preceding JSON object:

`Dict[str, Any]`
: The values may be of any type.

`Dict[str, Union[str, int, List[str]]]`
: Hard to read, and doesn't preserve the relationship between field names and their respective field types: `title` is supposed to be a `str`, it can't be an `int` or a `List[str]`.

PEP 589—TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys addressed that problem. Example 15-4 shows a simple `TypedDict`.

Example 15-4. books.py: the `BookDict` definition

```
from typing import TypedDict

class BookDict(TypedDict):
    isbn: str
    title: str
    authors: list[str]
    pagecount: int
```

At first glance, `typing.TypedDict` may seem like a data class builder, similar to `typing.NamedTuple`—covered in Chapter 5.

The syntactic similarity is misleading. `TypedDict` is very different. It exists only for the benefit of type checkers, and has no runtime effect.

`TypedDict` provides two things:

- Class-like syntax to annotate a `dict` with type hints for the value of each "field."
- A constructor that tells the type checker to expect a `dict` with the keys and values as specified.

At runtime, a `TypedDict` constructor such as `BookDict` is a placebo: it has the same effect as calling the `dict` constructor with the same arguments.

The fact that `BookDict` creates a plain `dict` also means that:

- The "fields" in the pseudoclass definition don't create instance attributes.
- You can't write initializers with default values for the "fields."
- Method definitions are not allowed.

Let's explore the behavior of a `BookDict` at runtime (Example 15-5).

Example 15-5. Using a `BookDict`, but not quite as intended

```
>>> from books import BookDict
>>> pp = BookDict(title='Programming Pearls',  ❶
...               authors='Jon Bentley',  ❷
```

```
...               isbn='0201657880',
...               pagecount=256)
>>> pp  ❸
{'title': 'Programming Pearls', 'authors': 'Jon Bentley', 'isbn': '0201657880',
 'pagecount': 256}
>>> type(pp)
<class 'dict'>
>>> pp.title  ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'title'
>>> pp['title']
'Programming Pearls'
>>> BookDict.__annotations__  ❺
{'isbn': <class 'str'>, 'title': <class 'str'>, 'authors': typing.List[str],
 'pagecount': <class 'int'>}
```

❶ You can call `BookDict` like a `dict` constructor with keyword arguments, or passing a `dict` argument—including a `dict` literal.

❷ Oops…I forgot `authors` takes a list. But gradual typing means no type checking at runtime.

❸ The result of calling `BookDict` is a plain `dict`…

❹ …therefore you can't read the data using `object.field` notation.

❺ The type hints are in `BookDict.__annotations__`, and not in `pp`.

Without a type checker, `TypedDict` is as useful as comments: it may help people read the code, but that's it. In contrast, the class builders from Chapter 5 are useful even if you don't use a type checker, because at runtime they generate or enhance a custom class that you can instantiate. They also provide several useful methods or functions listed in Table 5-1.

Example 15-6 builds a valid `BookDict` and tries some operations on it. This shows how `TypedDict` enables Mypy to catch errors, shown in Example 15-7.

Example 15-6. demo_books.py: legal and illegal operations on a `BookDict`

```
from books import BookDict
from typing import TYPE_CHECKING

def demo() -> None:  ❶
    book = BookDict(  ❷
        isbn='0134757599',
        title='Refactoring, 2e',
        authors=['Martin Fowler', 'Kent Beck'],
```

```
        pagecount=478
    )
    authors = book['authors'] ❸
    if TYPE_CHECKING:  ❹
        reveal_type(authors)  ❺
    authors = 'Bob'  ❻
    book['weight'] = 4.2
    del book['title']


if __name__ == '__main__':
    demo()
```

❶ Remember to add a return type, so that Mypy doesn't ignore the function.

❷ This is a valid `BookDict`: all the keys are present, with values of the correct types.

❸ Mypy will infer the type of `authors` from the annotation for the `'authors'` key in `BookDict`.

❹ `typing.TYPE_CHECKING` is only `True` when the program is being type checked. At runtime, it's always false.

❺ The previous `if` statement prevents `reveal_type(authors)` from being called at runtime. `reveal_type` is not a runtime Python function, but a debugging facility provided by Mypy. That's why there is no `import` for it. See its output in Example 15-7.

❻ The last three lines of the `demo` function are illegal. They will cause error messages in Example 15-7.

Type checking *demo_books.py* from Example 15-6, we get Example 15-7.

Example 15-7. Type checking demo_books.py

```
…/typeddict/ $ mypy demo_books.py
demo_books.py:13: note: Revealed type is 'built-ins.list[built-ins.str]'  ❶
demo_books.py:14: error: Incompatible types in assignment
                  (expression has type "str", variable has type "List[str]")  ❷
demo_books.py:15: error: TypedDict "BookDict" has no key 'weight'  ❸
demo_books.py:16: error: Key 'title' of TypedDict "BookDict" cannot be deleted  ❹
Found 3 errors in 1 file (checked 1 source file)
```

❶ This note is the result of `reveal_type(authors)`.

❷ The type of the `authors` variable was inferred from the type of the `book['authors']` expression that initialized it. You can't assign a `str` to a variable of type `List[str]`. Type checkers usually don't allow the type of a variable to change.[3]

❸ Cannot assign to a key that is not part of the `BookDict` definition.

❹ Cannot delete a key that is part of the `BookDict` definition.

Now let's see `BookDict` used in function signatures, to type check function calls.

Imagine you need to generate XML from book records, similar to this:

```
<BOOK>
  <ISBN>0134757599</ISBN>
  <TITLE>Refactoring, 2e</TITLE>
  <AUTHOR>Martin Fowler</AUTHOR>
  <AUTHOR>Kent Beck</AUTHOR>
  <PAGECOUNT>478</PAGECOUNT>
</BOOK>
```

If you were writing MicroPython code to be embedded in a tiny microcontroller, you might write a function like what's shown in Example 15-8.[4]

Example 15-8. books.py: `to_xml` function

```
AUTHOR_ELEMENT = '<AUTHOR>{}</AUTHOR>'

def to_xml(book: BookDict) -> str:  ❶
    elements: list[str] = []  ❷
    for key, value in book.items():
        if isinstance(value, list):  ❸
            elements.extend(
                AUTHOR_ELEMENT.format(n) for n in value)  ❹
        else:
            tag = key.upper()
            elements.append(f'<{tag}>{value}</{tag}>')
    xml = '\n\t'.join(elements)
    return f'<BOOK>\n\t{xml}\n</BOOK>'
```

3 As of May 2020, pytype allows it. But its FAQ says it will be disallowed in the future. See the question, "Why didn't pytype catch that I changed the type of an annotated variable?" in the pytype FAQ.

4 I prefer to use the lxml package to generate and parse XML: it's easy to get started, full-featured, and fast. Unfortunately, lxml and Python's own *ElementTree* don't fit the limited RAM of my hypothetical microcontroller.

❶ The whole point of the example: using `BookDict` in the function signature.

❷ It's often necessary to annotate collections that start empty, otherwise Mypy can't infer the type of the elements.[5]

❸ Mypy understands `isinstance` checks, and treats `value` as a `list` in this block.

❹ When I used `key == 'authors'` as the condition for the `if` guarding this block, Mypy found an error in this line: `"object" has no attribute "__iter__"`, because it inferred the type of `value` returned from `book.items()` as `object`, which doesn't support the `__iter__` method required by the generator expression. With the `isinstance` check, this works because Mypy knows that `value` is a `list` in this block.

Example 15-9 shows a function that parses a JSON `str` and returns a `BookDict`.

Example 15-9. books_any.py: `from_json` function

```
def from_json(data: str) -> BookDict:
    whatever = json.loads(data)  ❶
    return whatever  ❷
```

❶ The return type of `json.loads()` is `Any`.[6]

❷ I can return `whatever`—of type `Any`—because `Any` is *consistent-with* every type, including the declared return type, `BookDict`.

The second point of Example 15-9 is very important to keep in mind: Mypy will not flag any problem in this code, but at runtime the value in `whatever` may not conform to the `BookDict` structure—in fact, it may not be a `dict` at all!

If you run Mypy with `--disallow-any-expr`, it will complain about the two lines in the body of `from_json`:

```
…/typeddict/ $ mypy books_any.py --disallow-any-expr
books_any.py:30: error: Expression has type "Any"
books_any.py:31: error: Expression has type "Any"
Found 2 errors in 1 file (checked 1 source file)
```

5 The Mypy documentation discusses this in its "Common issues and solutions" page, in the section, "Types of empty collections".

6 Brett Cannon, Guido van Rossum, and others have been discussing how to type hint `json.loads()` since 2016 in Mypy issue #182: Define a JSON type.

Lines 30 and 31 mentioned in the previous snippet are the body of the `from_json` function. We can silence the type error by adding a type hint to the initialization of the `whatever` variable, as in Example 15-10.

Example 15-10. books.py: `from_json` function with variable annotation

```
def from_json(data: str) -> BookDict:
    whatever: BookDict = json.loads(data)  ❶
    return whatever  ❷
```

❶ `--disallow-any-expr` does not cause errors when an expression of type `Any` is immediately assigned to a variable with a type hint.

❷ Now `whatever` is of type `BookDict`, the declared return type.

Don't be lulled into a false sense of type safety by Example 15-10! Looking at the code at rest, the type checker cannot predict that `json.loads()` will return anything that resembles a `BookDict`. Only runtime validation can guarantee that.

Static type checking is unable to prevent errors with code that is inherently dynamic, such as `json.loads()`, which builds Python objects of different types at runtime, as Examples 15-11, 15-12, and 15-13 demonstrate.

Example 15-11. demo_not_book.py: `from_json` returns an invalid `BookDict`, and `to_xml` accepts it

```
from books import to_xml, from_json
from typing import TYPE_CHECKING

def demo() -> None:
    NOT_BOOK_JSON = """
        {"title": "Andromeda Strain",
         "flavor": "pistachio",
         "authors": true}
    """
    not_book = from_json(NOT_BOOK_JSON)  ❶
    if TYPE_CHECKING:  ❷
        reveal_type(not_book)
        reveal_type(not_book['authors'])

    print(not_book)  ❸
    print(not_book['flavor'])  ❹

    xml = to_xml(not_book)  ❺
    print(xml)  ❻
```

```
if __name__ == '__main__':
    demo()
```

❶ This line does not produce a valid `BookDict`—see the content of `NOT_BOOK_JSON`.

❷ Let's have Mypy reveal a couple of types.

❸ This should not be a problem: `print` can handle `object` and every other type.

❹ `BookDict` has no `'flavor'` key, but the JSON source does…what will happen?

❺ Remember the signature: `def to_xml(book: BookDict) -> str:`.

❻ What will the XML output look like?

Now we check *demo_not_book.py* with Mypy (Example 15-12).

Example 15-12. Mypy report for demo_not_book.py, reformatted for clarity

```
…/typeddict/ $ mypy demo_not_book.py
demo_not_book.py:12: note: Revealed type is
   'TypedDict('books.BookDict', {'isbn': built-ins.str,
                                  'title': built-ins.str,
                                  'authors': built-ins.list[built-ins.str],
                                  'pagecount': built-ins.int})'  ❶
demo_not_book.py:13: note: Revealed type is 'built-ins.list[built-ins.str]'  ❷
demo_not_book.py:16: error: TypedDict "BookDict" has no key 'flavor'  ❸
Found 1 error in 1 file (checked 1 source file)
```

❶ The revealed type is the nominal type, not the runtime content of `not_book`.

❷ Again, this is the nominal type of `not_book['authors']`, as defined in `BookDict`. Not the runtime type.

❸ This error is for line `print(not_book['flavor'])`: that key does not exist in the nominal type.

Now let's run *demo_not_book.py*, showing the output in Example 15-13.

Example 15-13. Output of running `demo_not_book.py`

```
…/typeddict/ $ python3 demo_not_book.py
{'title': 'Andromeda Strain', 'flavor': 'pistachio', 'authors': True}  ❶
pistachio  ❷
<BOOK>  ❸
```

```
        <TITLE>Andromeda Strain</TITLE>
        <FLAVOR>pistachio</FLAVOR>
        <AUTHORS>True</AUTHORS>
</BOOK>
```

❶ This is not really a `BookDict`.

❷ The value of `not_book['flavor']`.

❸ `to_xml` takes a `BookDict` argument, but there is no runtime checking: garbage in, garbage out.

Example 15-13 shows that *demo_not_book.py* outputs nonsense, but has no runtime errors. Using a `TypedDict` while handling JSON data did not provide much type safety.

If you look at the code for `to_xml` in Example 15-8 through the lens of duck typing, the argument `book` must provide an `.items()` method that returns an iterable of tuples like `(key, value)` where:

- `key` must have an `.upper()` method
- `value` can be anything

The point of this demonstration: when handling data with a dynamic structure, such as JSON or XML, `TypedDict` is absolutely not a replacement for data validation at runtime. For that, use *pydantic*.

`TypedDict` has more features, including support for optional keys, a limited form of inheritance, and an alternative declaration syntax. If you want to know more about it, please review PEP 589—TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys.

Now let's turn our attention to a function that is best avoided, but sometimes is unavoidable: `typing.cast`.

Type Casting

No type system is perfect, and neither are the static type checkers, the type hints in the *typeshed* project, or the type hints in the third-party packages that have them.

The `typing.cast()` special function provides one way to handle type checking malfunctions or incorrect type hints in code we can't fix. The Mypy 0.930 documentation explains:

> Casts are used to silence spurious type checker warnings and give the type checker a little help when it can't quite understand what is going on.

At runtime, `typing.cast` does absolutely nothing. This is its implementation:

```
def cast(typ, val):
    """Cast a value to a type.
    This returns the value unchanged.  To the type checker this
    signals that the return value has the designated type, but at
    runtime we intentionally don't check anything (we want this
    to be as fast as possible).
    """
    return val
```

PEP 484 requires type checkers to "blindly believe" the type stated in the `cast`. The "Casts" section of PEP 484 gives an example where the type checker needs the guidance of `cast`:

```
from typing import cast

def find_first_str(a: list[object]) -> str:
    index = next(i for i, x in enumerate(a) if isinstance(x, str))
    # We only get here if there's at least one string
    return cast(str, a[index])
```

The `next()` call on the generator expression will either return the index of a `str` item or raise `StopIteration`. Therefore, `find_first_str` will always return a `str` if no exception is raised, and `str` is the declared return type.

But if the last line were just `return a[index]`, Mypy would infer the return type as `object` because the `a` argument is declared as `list[object]`. So the `cast()` is required to guide Mypy.[7]

Here is another example with `cast`, this time to correct an outdated type hint for Python's standard library. In Example 21-12, I create an *asyncio* `Server` object and I want to get the address the server is listening to. I coded this line:

```
addr = server.sockets[0].getsockname()
```

But Mypy reported this error:

```
Value of type "Optional[List[socket]]" is not indexable
```

The type hint for `Server.sockets` on *typeshed* in May 2021 is valid for Python 3.6, where the `sockets` attribute could be `None`. But in Python 3.7, `sockets` became a property with a getter that always returns a `list`—which may be empty if the server has no sockets. And since Python 3.8, the getter returns a `tuple` (used as an immutable sequence).

7 The use of `enumerate` in the example is intended to confuse the type checker. A simpler implementation yielding strings directly instead of going through the `enumerate` index is correctly analyzed by Mypy, and the `cast()` is not needed.

Since I can't fix *typeshed* right now,[8] I added a `cast`, like this:

```
from asyncio.trsock import TransportSocket
from typing import cast

# ... many lines omitted ...

    socket_list = cast(tuple[TransportSocket, ...], server.sockets)
    addr = socket_list[0].getsockname()
```

Using `cast` in this case required a couple of hours to understand the problem and read *asyncio* source code to find the correct type of the sockets: the `TransportSocket` class from the undocumented `asyncio.trsock` module. I also had to add two `import` statements and another line of code for readability.[9] But the code is safer.

The careful reader may note that `sockets[0]` could raise `IndexError` if `sockets` is empty. However, as far as I understand `asyncio`, that cannot happen in Example 21-12 because the `server` is ready to accept connections by the time I read its `sockets` attribute, therefore it will not be empty. Anyway, `IndexError` is a runtime error. Mypy can't spot the problem even in a trivial case like `print([][0])`.

Don't get too comfortable using `cast` to silence Mypy, because Mypy is usually right when it reports an error. If you are using `cast` very often, that's a code smell. Your team may be misusing type hints, or you may have low-quality dependencies in your codebase.

Despite the downsides, there are valid uses for `cast`. Here is something Guido van Rossum wrote about it:

> What's wrong with the occasional `cast()` call or `# type: ignore` comment?[10]

8 I reported *typeshed* issue #5535, "Wrong type hint for asyncio.base_events.Server sockets attribute." and it was quickly fixed by Sebastian Rittau. However, I decided to keep the example because it illustrates a common use case for `cast`, and the `cast` I wrote is harmless.

9 To be honest, I originally appended a `# type: ignore` comment to the line with `server.sockets[0]` because after a little research I found similar lines the *asyncio* documentation and in a test case, so I suspected the problem was not in my code.

10 19 May 2020 message to the typing-sig mailing list.

It is unwise to completely ban the use of `cast`, especially because the other workarounds are worse:

- `# type: ignore` is less informative.[11]
- Using `Any` is contagious: since `Any` is *consistent-with* all types, abusing it may produce cascading effects through type inference, undermining the type checker's ability to detect errors in other parts of the code.

Of course, not all typing mishaps can be fixed with `cast`. Sometimes we need `# type: ignore`, the occasional `Any`, or even leaving a function without type hints.

Next, let's talk about using annotations at runtime.

Reading Type Hints at Runtime

At import time, Python reads the type hints in functions, classes, and modules, and stores them in attributes named `__annotations__`. For instance, consider the `clip` function in Example 15-14.[12]

Example 15-14. clipannot.py: annotated signature of a `clip` function

```
def clip(text: str, max_len: int = 80) -> str:
```

The type hints are stored as a `dict` in the `__annotations__` attribute of the function:

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

The `'return'` key maps to the return type hint after the `->` symbol in Example 15-14.

Note that the annotations are evaluated by the interpreter at import time, just as parameter default values are also evaluated. That's why the values in the annotations are the Python classes `str` and `int`, and not the strings `'str'` and `'int'`. The import time evaluation of annotations is the standard as of Python 3.10, but that may change if PEP 563 or PEP 649 become the standard behavior.

11 The syntax `# type: ignore[code]` allows you to specify which Mypy error code is being silenced, but the codes are not always easy to interpret. See "Error codes" in the Mypy documentation.

12 I will not go into the implementation of `clip`, but you can read the whole module in *clip_annot.py* if you're curious.

Problems with Annotations at Runtime

The increased use of type hints raised two problems:

- Importing modules uses more CPU and memory when many type hints are used.
- Referring to types not yet defined requires using strings instead of actual types.

Both issues are relevant. The first is because of what we just saw: annotations are evaluated by the interpreter at import time and stored in the `__annotations__` attribute. Let's focus now on the second issue.

Storing annotations as strings is sometimes required because of the "forward reference" problem: when a type hint needs to refer to a class defined below in the same module. However, a common manifestation of the problem in source code doesn't look like a forward reference at all: that's when a method returns a new object of the same class. Since the class object is not defined until Python completely evaluates the class body, type hints must use the name of the class as a string. Here is an example:

```
class Rectangle:
    # ... lines omitted ...
    def stretch(self, factor: float) -> 'Rectangle':
        return Rectangle(width=self.width * factor)
```

Writing forward referencing type hints as strings is the standard and required practice as of Python 3.10. Static type checkers were designed to deal with that issue from the beginning.

But at runtime, if you write code to read the `return` annotation for `stretch`, you will get a string `'Rectangle'` instead of a reference to the actual type, the `Rectangle` class. Now your code needs to figure out what that string means.

The `typing` module includes three functions and a class categorized as Introspection helpers, the most important being `typing.get_type_hints`. Part of its documentation states:

`get_type_hints(obj, globals=None, locals=None, include_extras=False)`
> […] This is often the same as `obj.__annotations__`. In addition, forward references encoded as string literals are handled by evaluating them in `globals` and `locals` namespaces. […]

Since Python 3.10, the new `inspect.get_annotations(…)` function should be used instead of `typing.get_type_hints`. However, some readers may not be using Python 3.10 yet, so in the examples I'll use `typing.get_type_hints`, which is available since the `typing` module was added in Python 3.5.

PEP 563—Postponed Evaluation of Annotations was approved to make it unnecessary to write annotations as strings, and to reduce the runtime costs of type hints. Its main idea is described in these two sentences of the "Abstract":

> This PEP proposes changing function annotations and variable annotations so that they are no longer evaluated at function definition time. Instead, they are preserved in *annotations* in string form.

Beginning with Python 3.7, that's how annotations are handled in any module that starts with this `import` statement:

```
from __future__ import annotations
```

To demonstrate its effect, I put a copy of the same `clip` function from Example 15-14 in a *clip_annot_post.py* module with that `__future__` import line at the top.

At the console, here's what I get when I import that module and read the annotations from `clip`:

```
>>> from clip_annot_post import clip
>>> clip.__annotations__
{'text': 'str', 'max_len': 'int', 'return': 'str'}
```

As you can see, all the type hints are now plain strings, despite the fact they are not written as quoted strings in the definition of `clip` (Example 15-14).

The `typing.get_type_hints` function is able to resolve many type hints, including those in `clip`:

```
>>> from clip_annot_post import clip
>>> from typing import get_type_hints
>>> get_type_hints(clip)
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

Calling `get_type_hints` gives us the real types—even in some cases where the original type hint is written as a quoted string. That's the recommended way to read type hints at runtime.

The PEP 563 behavior was scheduled to become default in Python 3.10, with no `__future__` import needed. However, the maintainers of *FastAPI* and *pydantic* raised the alarm that the change would break their code which relies on type hints at runtime, and cannot use `get_type_hints` reliably.

In the ensuing discussion on the python-dev mailing list, Łukasz Langa—the author of PEP 563—described some limitations of that function:

> […] it turned out that `typing.get_type_hints()` has limits that make its use in general costly at runtime, and more importantly insufficient to resolve all types. The most common example deals with non-global context in which types are generated (e.g., inner classes, classes within functions, etc.). But one of the crown examples of forward references: classes with methods accepting or returning objects of their own type, also

isn't properly handled by `typing.get_type_hints()` if a class generator is used. There's some trickery we can do to connect the dots but in general it's not great.[13]

Python's Steering Council decided to postpone making PEP 563 the default behavior until Python 3.11 or later, giving more time to developers to come up with a solution that addresses the issues PEP 563 tried to solve, without breaking widespread uses of type hints at runtime. PEP 649—Deferred Evaluation Of Annotations Using Descriptors is under consideration as a possible solution, but a different compromise may be reached.

To summarize: reading type hints at runtime is not 100% reliable as of Python 3.10 and is likely to change in 2022.

Companies using Python at a very large scale want the benefits of static typing, but they don't want to pay the price for the evaluation of the type hints at import time. Static checking happens at developer workstations and dedicated CI servers, but loading modules happens at a much higher frequency and volume in the production containers, and this cost is not negligible at scale.

This creates tension in the Python community between those who want type hints to be stored as strings only—to reduce the loading costs—versus those who also want to use type hints at runtime, like the creators and users of *pydantic* and *FastAPI*, who would rather have type objects stored instead of having to evaluate those annotations, a challenging task.

Dealing with the Problem

Given the unstable situation at present, if you need to read annotations at runtime, I recommend:

- Avoid reading `__annotations__` directly; instead, use `inspect.get_annotations` (from Python 3.10) or `typing.get_type_hints` (since Python 3.5).
- Write a custom function of your own as a thin wrapper around `inspect.get_annotations` or `typing.get_type_hints`, and have the rest of your codebase call that custom function, so that future changes are localized to a single function.

To demonstrate the second point, here are the first lines of the `Checked` class defined in Example 24-5, which we'll study in Chapter 24:

13 Message "PEP 563 in light of PEP 649", posted April 16, 2021.

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)
    # ... more lines ...
```

The `Checked._fields` class method protects other parts of the module from depending directly on `typing.get_type_hints`. If `get_type_hints` changes in the future, requiring additional logic, or you want to replace it with `inspect.get_annotations`, the change is limited to `Checked._fields` and does not affect the rest of your program.

Given the ongoing discussions and proposed changes for runtime inspection of type hints, the official "Annotations Best Practices" document is required reading, and is likely to be updated on the road to Python 3.11. That how-to was written by Larry Hastings, the author of PEP 649—Deferred Evaluation Of Annotations Using Descriptors, an alternative proposal to address the runtime issues raised by PEP 563—Postponed Evaluation of Annotations.

The remaining sections of this chapter cover generics, starting with how to define a generic class that can be parameterized by its users.

Implementing a Generic Class

In Example 13-7 we defined the `Tombola` ABC: an interface for classes that work like a bingo cage. The `LottoBlower` class from Example 13-10 is a concrete implementation. Now we'll study a generic version of `LottoBlower` used like in Example 15-15.

Example 15-15. generic_lotto_demo.py: using a generic lottery blower class

```
from generic_lotto import LottoBlower

machine = LottoBlower[int](range(1, 11))  ❶

first = machine.pick()  ❷
remain = machine.inspect()  ❸
```

❶ To instantiate a generic class, we give it an actual type parameter, like `int` here.

❷ Mypy will correctly infer that `first` is an `int`…

❸ … and that `remain` is a `tuple` of integers.

In addition, Mypy reports violations of the parameterized type with helpful messages, such as what's shown in Example 15-16.

Example 15-16. generic_lotto_errors.py: errors reported by Mypy

```
from generic_lotto import LottoBlower

machine = LottoBlower[int]([1, .2])
## error: List item 1 has incompatible type "float";  ❶
##        expected "int"

machine = LottoBlower[int](range(1, 11))

machine.load('ABC')
## error: Argument 1 to "load" of "LottoBlower"  ❷
##        has incompatible type "str";
##        expected "Iterable[int]"
## note:  Following member(s) of "str" have conflicts:
## note:      Expected:
## note:          def __iter__(self) -> Iterator[int]
## note:      Got:
## note:          def __iter__(self) -> Iterator[str]
```

❶ Upon instantiation of `LottoBlower[int]`, Mypy flags the `float`.

❷ When calling `.load('ABC')`, Mypy explains why a `str` won't do: `str.__iter__` returns an `Iterator[str]`, but `LottoBlower[int]` requires an `Iterator[int]`.

Example 15-17 is the implementation.

Example 15-17. generic_lotto.py: a generic lottery blower class

```
import random

from collections.abc import Iterable
from typing import TypeVar, Generic

from tombola import Tombola

T = TypeVar('T')

class LottoBlower(Tombola, Generic[T]):  ❶

    def __init__(self, items: Iterable[T]) -> None:  ❷
        self._balls = list[T](items)

    def load(self, items: Iterable[T]) -> None:  ❸
        self._balls.extend(items)
```

```
    def pick(self) -> T:  ❹
        try:
            position = random.randrange(len(self._balls))
        except ValueError:
            raise LookupError('pick from empty LottoBlower')
        return self._balls.pop(position)

    def loaded(self) -> bool:  ❺
        return bool(self._balls)

    def inspect(self) -> tuple[T, ...]:  ❻
        return tuple(self._balls)
```

❶ Generic class declarations often use multiple inheritance, because we need to subclass `Generic` to declare the formal type parameters—in this case, `T`.

❷ The `items` argument in `__init__` is of type `Iterable[T]`, which becomes `Iterable[int]` when an instance is declared as `LottoBlower[int]`.

❸ The `load` method is likewise constrained.

❹ The return type of `T` now becomes `int` in a `LottoBlower[int]`.

❺ No type variable here.

❻ Finally, `T` sets the type of the items in the returned `tuple`.

The "User-defined generic types" section of the `typing` module documentation is short, presents good examples, and provides a few more details that I do not cover here.

Now that we've seen how to implement a generic class, let's define the terminology to talk about generics.

Basic Jargon for Generic Types

Here are a few definitions that I found useful when studying generics:[14]

Generic type
: A type declared with one or more type variables.
Examples: `LottoBlower[T]`, `abc.Mapping[KT, VT]`

Formal type parameter
: The type variables that appear in a generic type declaration.
Example: `KT` and `VT` in the previous example `abc.Mapping[KT, VT]`

Parameterized type
: A type declared with actual type parameters.
Examples: `LottoBlower[int]`, `abc.Mapping[str, float]`

Actual type parameter
: The actual types given as parameters when a parameterized type is declared.
Example: the `int` in `LottoBlower[int]`

The next topic is about how to make generic types more flexible, introducing the concepts of covariance, contravariance, and invariance.

Variance

Depending on your experience with generics in other languages, this may be the most challenging section in the book. The concept of variance is abstract, and a rigorous presentation would make this section look like pages from a math book.

In practice, variance is mostly relevant to library authors who want to support new generic container types or provide callback-based APIs. Even then, you can avoid much complexity by supporting only invariant containers—which is mostly what we have now in the Python standard library. So, on a first reading, you can skip the whole section or just read the sections about invariant types.

We first saw the concept of *variance* in "Variance in Callable types" on page 292, applied to parameterized generic `Callable` types. Here we'll expand the concept to cover generic collection types, using a "real world" analogy to make this abstract concept more concrete.

14 The terms are from Joshua Bloch's classic book, *Effective Java*, 3rd ed. (Addison-Wesley). The definitions and examples are mine.

Imagine that a school cafeteria has a rule that only juice dispensers can be installed.[15] General beverage dispensers are not allowed because they may serve sodas, which are banned by the school board.[16]

An Invariant Dispenser

Let's try to model the cafeteria scenario with a generic `BeverageDispenser` class that can be parameterized on the type of beverage. See Example 15-18.

Example 15-18. invariant.py: type definitions and `install` function

```
from typing import TypeVar, Generic

class Beverage:  ❶
    """Any beverage."""

class Juice(Beverage):
    """Any fruit juice."""

class OrangeJuice(Juice):
    """Delicious juice from Brazilian oranges."""

T = TypeVar('T')  ❷

class BeverageDispenser(Generic[T]):  ❸
    """A dispenser parameterized on the beverage type."""
    def __init__(self, beverage: T) -> None:
        self.beverage = beverage

    def dispense(self) -> T:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None:  ❹
    """Install a fruit juice dispenser."""
```

❶ `Beverage`, `Juice`, and `OrangeJuice` form a type hierarchy.

❷ Simple `TypeVar` declaration.

❸ `BeverageDispenser` is parameterized on the type of beverage.

15 I first saw the cafeteria analogy for variance in Erik Meijer's *Foreword* in *The Dart Programming Language* book by Gilad Bracha (Addison-Wesley).

16 Much better than banning books!

❹ `install` is a module-global function. Its type hint enforces the rule that only a juice dispenser is acceptable.

Given the definitions in Example 15-18, the following code is legal:

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)
```

However, this is not legal:

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##          expected "BeverageDispenser[Juice]"
```

A dispenser that serves any `Beverage` is not acceptable because the cafeteria requires a dispenser that is specialized for `Juice`.

Somewhat surprisingly, this code is also illegal:

```
orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[OrangeJuice]"
##          expected "BeverageDispenser[Juice]"
```

A dispenser specialized for `OrangeJuice` is not allowed either. Only `BeverageDispenser[Juice]` will do. In the typing jargon, we say that `BeverageDispenser(Generic[T])` is invariant when `BeverageDispenser[OrangeJuice]` is not compatible with `BeverageDispenser[Juice]`—despite the fact that `OrangeJuice` is a *subtype-of* `Juice`.

Python mutable collection types—such as `list` and `set`—are invariant. The `LottoBlower` class from Example 15-17 is also invariant.

A Covariant Dispenser

If we want to be more flexible and model dispensers as a generic class that can accept some beverage type and also its subtypes, we must make it covariant. Example 15-19 shows how we'd declare `BeverageDispenser`.

Example 15-19. covariant.py: type definitions and `install` function

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


class BeverageDispenser(Generic[T_co]):  ❷
    def __init__(self, beverage: T_co) -> None:
        self.beverage = beverage
```

```
    def dispense(self) -> T_co:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None:  ❸
    """Install a fruit juice dispenser."""
```

❶ Set `covariant=True` when declaring the type variable; `_co` is a conventional suffix for covariant type parameters on *typeshed*.

❷ Use `T_co` to parameterize the `Generic` special class.

❸ Type hints for `install` are the same as in Example 15-18.

The following code works because now both the `Juice` dispenser and the `Orange Juice` dispenser are valid in a covariant `BeverageDispenser`:

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)

orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
```

But a dispenser for an arbitrary `Beverage` is not acceptable:

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##          expected "BeverageDispenser[Juice]"
```

That's covariance: the subtype relationship of the parameterized dispensers varies in the same direction as the subtype relationship of the type parameters.

A Contravariant Trash Can

Now we'll model the cafeteria rule for deploying a trash can. Let's assume food and drinks are served in biodegradable packages, and leftovers as well as single-use utensils are also biodegradable. The trash cans must be suitable for biodegradable refuse.

For the sake of this didactic example, let's make simplifying assumptions to classify trash in a neat hierarchy:

- `Refuse` is the most general type of trash. All trash is refuse.
- `Biodegradable` is a specific type of trash that can be decomposed by organisms over time. Some `Refuse` is not `Biodegradable`.
- `Compostable` is a specific type of `Biodegradable` trash that can be efficiently turned into organic fertilizer in a compost bin or in a composting facility. Not all `Biodegradable` trash is `Compostable` in our definition.

In order to model the rule for an acceptable trash can in the cafeteria, we need to introduce the concept of "contravariance" through an example using it, as shown in Example 15-20.

Example 15-20. contravariant.py: type definitions and `install` function

```
from typing import TypeVar, Generic

class Refuse:  ❶
    """Any refuse."""

class Biodegradable(Refuse):
    """Biodegradable refuse."""

class Compostable(Biodegradable):
    """Compostable refuse."""

T_contra = TypeVar('T_contra', contravariant=True)  ❷

class TrashCan(Generic[T_contra]):  ❸
    def put(self, refuse: T_contra) -> None:
        """Store trash until dumped."""

def deploy(trash_can: TrashCan[Biodegradable]):
    """Deploy a trash can for biodegradable refuse."""
```

❶ A type hierarchy for refuse: `Refuse` is the most general type, `Compostable` is the most specific.

❷ `T_contra` is a conventional name for a contravariant type variable.

❸ `TrashCan` is contravariant on the type of refuse.

Given those definitions, these types of trash cans are acceptable:

```
bio_can: TrashCan[Biodegradable] = TrashCan()
deploy(bio_can)

trash_can: TrashCan[Refuse] = TrashCan()
deploy(trash_can)
```

The more general `TrashCan[Refuse]` is acceptable because it can take any kind of refuse, including `Biodegradable`. However, a `TrashCan[Compostable]` will not do, because it cannot take `Biodegradable`:

```
compost_can: TrashCan[Compostable] = TrashCan()
deploy(compost_can)
## mypy: Argument 1 to "deploy" has
## incompatible type "TrashCan[Compostable]"
##          expected "TrashCan[Biodegradable]"
```

Let's summarize the concepts we just saw.

Variance Review

Variance is a subtle property. The following sections recap the concept of invariant, covariant, and contravariant types, and provide some rules of thumb to reason about them.

Invariant types

A generic type `L` is invariant when there is no supertype or subtype relationship between two parameterized types, regardless of the relationship that may exist between the actual parameters. In other words, if `L` is invariant, then `L[A]` is not a supertype or a subtype of `L[B]`. They are inconsistent in both ways.

As mentioned, Python's mutable collections are invariant by default. The `list` type is a good example: `list[int]` is not *consistent-with* `list[float]` and vice versa.

In general, if a formal type parameter appears in type hints of method arguments, and the same parameter appears in method return types, that parameter must be invariant to ensure type safety when updating and reading from the collection.

For example, here is part of the type hints for the `list` built-in on *typeshed*:

```
class list(MutableSequence[_T], Generic[_T]):
    @overload
    def __init__(self) -> None: ...
    @overload
    def __init__(self, iterable: Iterable[_T]) -> None: ...
    # ... lines omitted ...
    def append(self, __object: _T) -> None: ...
    def extend(self, __iterable: Iterable[_T]) -> None: ...
    def pop(self, __index: int = ...) -> _T: ...
    # etc...
```

Note that `_T` appears in the arguments of `__init__`, `append`, and `extend`, and as the return type of `pop`. There is no way to make such a class type safe if it is covariant or contravariant in `_T`.

Covariant types

Consider two types `A` and `B`, where `B` is *consistent-with* `A`, and neither of them is `Any`. Some authors use the `<:` and `:>` symbols to denote type relationships like this:

`A :> B`
: `A` is a *supertype-of* or the same as `B`.

`B <: A`
: `B` is a *subtype-of* or the same as `A`.

Given `A :> B`, a generic type `C` is covariant when `C[A] :> C[B]`.

Note the direction of the `:>` symbol is the same in both cases where `A` is to the left of `B`. Covariant generic types follow the subtype relationship of the actual type parameters.

Immutable containers can be covariant. For example, this is how the `typing.FrozenSet` class is documented as a covariant with a type variable using the conventional name `T_co`:

```
class FrozenSet(frozenset, AbstractSet[T_co]):
```

Applying the `:>` notation to parameterized types, we have:

```
             float :> int
frozenset[float] :> frozenset[int]
```

Iterators are another example of covariant generics: they are not read-only collections like a `frozenset`, but they only produce output. Any code expecting an `abc.Iterator[float]` yielding floats can safely use an `abc.Iterator[int]` yielding integers. `Callable` types are covariant on the return type for a similar reason.

Contravariant types

Given `A :> B`, a generic type `K` is contravariant if `K[A] <: K[B]`.

Contravariant generic types reverse the subtype relationship of the actual type parameters.

The `TrashCan` class exemplifies this:

```
            Refuse :> Biodegradable
TrashCan[Refuse] <: TrashCan[Biodegradable]
```

A contravariant container is usually a write-only data structure, also known as a "sink." There are no examples of such collections in the standard library, but there are a few types with contravariant type parameters.

`Callable[[ParamType, …], ReturnType]` is contravariant on the parameter types, but covariant on the `ReturnType`, as we saw in "Variance in Callable types" on page 292. In addition, `Generator`, `Coroutine`, and `AsyncGenerator` have one contravariant type parameter. The `Generator` type is described in "Generic Type Hints for Classic Coroutines" on page 650; `Coroutine` and `AsyncGenerator` are described in Chapter 21.

For the present discussion about variance, the main point is that the contravariant formal parameter defines the type of the arguments used to invoke or send data to the object, while different covariant formal parameters define the types of outputs produced by the object—the yield type or the return type, depending on the object. The meanings of "send" and "yield" are explained in "Classic Coroutines" on page 641.

We can derive useful guidelines from these observations of covariant outputs and contravariant inputs.

Variance rules of thumb

Finally, here are a few rules of thumb to reason about when thinking through variance:

- If a formal type parameter defines a type for data that comes out of the object, it can be covariant.
- If a formal type parameter defines a type for data that goes into the object after its initial construction, it can be contravariant.
- If a formal type parameter defines a type for data that comes out of the object and the same parameter defines a type for data that goes into the object, it must be invariant.
- To err on the safe side, make formal type parameters invariant.

`Callable[[ParamType, …], ReturnType]` demonstrates rules #1 and #2: The `ReturnType` is covariant, and each `ParamType` is contravariant.

By default, `TypeVar` creates formal parameters that are invariant, and that's how the mutable collections in the standard library are annotated.

"Generic Type Hints for Classic Coroutines" on page 650 continues the present discussion about variance.

Next, let's see how to define generic static protocols, applying the idea of covariance to a couple of new examples.

Implementing a Generic Static Protocol

The Python 3.10 standard library provides a few generic static protocols. One of them is `SupportsAbs`, implemented like this in the *typing* module:

```
    @runtime_checkable
    class SupportsAbs(Protocol[T_co]):
        """An ABC with one abstract method __abs__ that is covariant in its
            return type."""
        __slots__ = ()

        @abstractmethod
        def __abs__(self) -> T_co:
            pass
```

`T_co` is declared according to the naming convention:

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

Thanks to `SupportsAbs`, Mypy recognizes this code as valid, as you can see in Example 15-21.

Example 15-21. abs_demo.py: use of the generic `SupportsAbs` protocol

```
import math
from typing import NamedTuple, SupportsAbs

class Vector2d(NamedTuple):
    x: float
    y: float

    def __abs__(self) -> float:  ❶
        return math.hypot(self.x, self.y)

def is_unit(v: SupportsAbs[float]) -> bool:  ❷
    """'True' if the magnitude of 'v' is close to 1."""
    return math.isclose(abs(v), 1.0)  ❸

assert issubclass(Vector2d, SupportsAbs)  ❹

v0 = Vector2d(0, 1)  ❺
sqrt2 = math.sqrt(2)
v1 = Vector2d(sqrt2 / 2, sqrt2 / 2)
v2 = Vector2d(1, 1)
v3 = complex(.5, math.sqrt(3) / 2)
v4 = 1  ❻

assert is_unit(v0)
assert is_unit(v1)
assert not is_unit(v2)
assert is_unit(v3)
```

```
assert is_unit(v4)

print('OK')
```

❶ Defining `__abs__` makes `Vector2d` *consistent-with* `SupportsAbs`.

❷ Parameterizing `SupportsAbs` with `float` ensures…

❸ …that Mypy accepts `abs(v)` as the first argument for `math.isclose`.

❹ Thanks to `@runtime_checkable` in the definition of `SupportsAbs`, this is a valid runtime assertion.

❺ The remaining code all passes Mypy checks and runtime assertions.

❻ The `int` type is also *consistent-with* `SupportsAbs`. According to *typeshed*, `int.__abs__` returns an `int`, which is *consistent-with* the `float` type parameter declared in the `is_unit` type hint for the `v` argument.

Similarly, we can write a generic version of the `RandomPicker` protocol presented in Example 13-18, which was defined with a single method `pick` returning `Any`.

Example 15-22 shows how to make a generic `RandomPicker` covariant on the return type of `pick`.

Example 15-22. generic_randompick.py: definition of generic `RandomPicker`

```
from typing import Protocol, runtime_checkable, TypeVar

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

@runtime_checkable
class RandomPicker(Protocol[T_co]):  ❷
    def pick(self) -> T_co: ...  ❸
```

❶ Declare `T_co` as `covariant`.

❷ This makes `RandomPicker` generic with a covariant formal type parameter.

❸ Use `T_co` as the return type.

The generic `RandomPicker` protocol can be covariant because its only formal parameter is used in a return type.

With this, we can call it a chapter.

Chapter Summary

The chapter started with a simple example of using `@overload`, followed by a much more complex example that we studied in detail: the overloaded signatures required to correctly annotate the `max` built-in function.

The `typing.TypedDict` special construct came next. I chose to cover it here, and not in Chapter 5 where we saw `typing.NamedTuple`, because `TypedDict` is not a class builder; it's merely a way to add type hints to a variable or argument that requires a `dict` with a specific set of string keys, and specific types for each key—which happens when we use a `dict` as a record, often in the context of handling with JSON data. That section was a bit long because using `TypedDict` can give a false sense of security, and I wanted to show how runtime checks and error handling are really inevitable when trying to make statically structured records out of mappings that are dynamic in nature.

Next we talked about `typing.cast`, a function designed to let us guide the work of the type checker. It's important to carefully consider when to use `cast`, because overusing it hinders the type checker.

Runtime access to type hints came next. The key point was to use `typing.get_type_hints` instead of reading the `__annotations__` attribute directly. However, that function may be unreliable with some annotations, and we saw that Python core developers are still working on a way to make type hints usable at runtime, while reducing their impact on CPU and memory usage.

The final sections were about generics, starting with the `LottoBlower` generic class—which we later learned is an invariant generic class. That example was followed by definitions of four basic terms: generic type, formal type parameter, parameterized type, and actual type parameter.

The major topic of variance was presented next, using cafeteria beverage dispensers and trash cans as "real life" examples of invariant, covariant, and contravariant generic types. Next we reviewed, formalized, and further applied those concepts to examples in Python's standard library.

Lastly, we saw how a generic static protocol is defined, first considering the `typing.SupportsAbs` protocol, and then applying the same idea to the `RandomPicker` example, making it more strict than the original protocol from Chapter 13.

Python's type system is a huge and rapidly evolving subject. This chapter is not comprehensive. I chose to focus on topics that are either widely applicable, particularly challenging, or conceptually important and therefore likely to be relevant for a long time.

Further Reading

Python's static type system was complex as initially designed, and is getting more complex with each passing year. Table 15-1 lists all the PEPs that I am aware of as of May 2021. It would take a whole book to cover everything.

*Table 15-1. PEPs about type hints, with links in the titles. PEP with numbers marked with * are important enough to be mentioned in the opening paragraph of the `typing` documentation. Question marks in the Python column indicate PEPs under discussion or not yet implemented; "n/a" appears in informational PEPs with no specific Python version.*

PEP	Title	Python	Year
3107	Function Annotations	3.0	2006
483*	The Theory of Type Hints	n/a	2014
484*	Type Hints	3.5	2014
482	Literature Overview for Type Hints	n/a	2015
526*	Syntax for Variable Annotations	3.6	2016
544*	Protocols: Structural subtyping (static duck typing)	3.8	2017
557	Data Classes	3.7	2017
560	Core support for typing module and generic types	3.7	2017
561	Distributing and Packaging Type Information	3.7	2017
563	Postponed Evaluation of Annotations	3.7	2017
586*	Literal Types	3.8	2018
585	Type Hinting Generics In Standard Collections	3.9	2019
589*	TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys	3.8	2019
591*	Adding a final qualifier to typing	3.8	2019
593	Flexible function and variable annotations	?	2019
604	Allow writing union types as X | Y	3.10	2019
612	Parameter Specification Variables	3.10	2019
613	Explicit Type Aliases	3.10	2020
645	Allow writing optional types as x?	?	2020
646	Variadic Generics	?	2020
647	User-Defined Type Guards	3.10	2021
649	Deferred Evaluation Of Annotations Using Descriptors	?	2021
655	Marking individual TypedDict items as required or potentially-missing	?	2021

Python's official documentation hardly keeps up with all that, so Mypy's documentation is an essential reference. *Robust Python* by Patrick Viafore (O'Reilly) is the first book with extensive coverage of Python's static type system that I know about, published August 2021. You may be reading the second such book right now.

The subtle topic of variance has its own section in PEP 484, and is also covered in the "Generics" page of Mypy, as well as in its invaluable "Common Issues" page.

PEP 362—Function Signature Object is worth reading if you intend to use the `inspect` module that complements the `typing.get_type_hints` function.

If you are interested in the history of Python, you may like to know that Guido van Rossum posted "Adding Optional Static Typing to Python" on December 23, 2004.

"Python 3 Types in the Wild: A Tale of Two Type Systems" is a research paper by Ingkarat Rak-amnouykit and others from the Rensselaer Polytechnic Institute and IBM TJ Watson Research Center. The paper surveys the use of type hints in open source projects on GitHub, showing that most projects don't use them, and also that most projects that have type hints apparently don't use a type checker. I found most interesting the discussion of the different semantics of Mypy and Google's *pytype*, which they conclude are "essentially two different type systems."

Two seminal papers about gradual typing are Gilad Bracha's "Pluggable Type Systems", and "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages" by Eric Meijer and Peter Drayton.[17]

I learned a lot reading the relevant parts of some books about other languages that implement some of the same ideas:

- *Atomic Kotlin* by Bruce Eckel and Svetlana Isakova (Mindview)
- *Effective Java*, 3rd ed., by Joshua Bloch (Addison-Wesley)
- *Programming with Types: TypeScript Examples* by Vlad Riscutia (Manning)
- *Programming TypeScript* by Boris Cherny (O'Reilly)
- *The Dart Programming Language* by Gilad Bracha (Addison-Wesley)[18]

For some critical views on type systems, I recommend Victor Youdaiken's posts "Bad ideas in type theory" and "Types considered harmful II".

17 As a reader of footnotes, you may recall that I credited Erik Meijer for the cafeteria analogy to explain variance.

18 That book was written for Dart 1. There are significant changes in Dart 2, including in the type system. Nevertheless, Bracha is an important researcher in the field of programming language design, and I found the book valuable for his perspective on the design of Dart.

Finally, I was surprised to find "Generics Considered Harmful" by Ken Arnold, a core contributor to Java from the beginning, as well as coauthor of the first four editions of the official *The Java Programming Language* book (Addison-Wesley)—in collaboration with James Gosling, the lead designer of Java.

Sadly, Arnold's criticism applies to Python's static type system as well. While reading the many rules and special cases of the typing PEPs, I was constantly reminded of this passage from Gosling's post:

> Which brings up the problem that I always cite for C++: I call it the "N^{th} order exception to the exception rule." It sounds like this: "You can do x, except in case y, unless y does z, in which case you can if …"

Fortunately, Python has a key advantage over Java and C++: an optional type system. We can squelch type checkers and omit type hints when they become too cumbersome.

Soapbox

Typing Rabbit Holes

When using a type checker, we are sometimes forced to discover and import classes we did not need to know about, and our code has no need to reference—except to write type hints. Such classes are undocumented, probably because they are considered implementation details by the authors of the packages. Here are two examples from the standard library.

To use `cast()` in the `server.sockets` example in "Type Casting" on page 534, I had to scour the vast *asyncio* documentation and then browse the source code of several modules in that package to discover the undocumented `TransportSocket` class in the equally undocumented `asyncio.trsock` module. Using `socket.socket` instead of `TransportSocket` would be incorrect, because the latter is explicitly not a subtype of the former, according to a docstring in the source code.

I fell into a similar rabbit hole when I added type hints to Example 19-13, a simple demonstration of `multiprocessing`. That example uses `SimpleQueue` objects, which you get by calling `multiprocessing.SimpleQueue()`. However, I could not use that name in a type hint, because it turns out that `multiprocessing.SimpleQueue` is not a class! It's a bound method of the undocumented `multiprocessing.BaseContext` class, which builds and returns an instance of the `SimpleQueue` class defined in the undocumented `multiprocessing.queues` module.

In each of those cases I had to spend a couple of hours to find the right undocumented class to import, just to write a single type hint. This kind of research is part of the job when writing a book. But when writing application code, I'd probably avoid such

scavenger hunts for a single offending line and just write `# type: ignore`. Sometimes that's the only cost-effective solution.

Variance Notation in Other Languages

Variance is a difficult topic, and Python's type hints syntax is not as good as it could be. This is evidenced by this direct quote from PEP 484:

> Covariance or contravariance is not a property of a type variable, but a property of a generic class defined using this variable.[19]

If that is the case, why are covariance and contravariance declared with `TypeVar` and not on the generic class?

The authors of PEP 484 worked under the severe self-imposed constraint that type hints should be supported without making any change to the interpreter. This required the introduction of `TypeVar` to define type variables, and also the abuse of `[]` to provide `Klass[T]` syntax for generics—instead of the `Klass<T>` notation used in other popular languages, including C#, Java, Kotlin, and TypeScript. None of these languages require type variables to be declared before use.

In addition, the syntax of Kotlin and C# makes it clear whether the type parameter is covariant, contravariant, or invariant exactly where it makes sense: in the class or interface declaration.

In Kotlin, we could declare the `BeverageDispenser` like this:

```
class BeverageDispenser<out T> {
    // etc...
}
```

The `out` modifier in the formal type parameter means `T` is an "output" type, therefore `BeverageDispenser` is covariant.

19 See the last paragraph of the section "Covariance and Contravariance" in PEP 484.

You can probably guess how `TrashCan` would be declared:

```
class TrashCan<in T> {
    // etc...
}
```

Given `T` as an "input" formal type parameter, then `TrashCan` is contravariant.

If neither `in` nor `out` appear, then the class is invariant on the parameter.

It's easy to recall the "Variance rules of thumb" on page 551 when `out` and `in` are used in the formal type parameters.

This suggests that a good naming convention for covariant and contravariant type variables in Python would be:

```
T_out = TypeVar('T_out', covariant=True)
T_in = TypeVar('T_in', contravariant=True)
```

Then we could define the classes like this:

```
class BeverageDispenser(Generic[T_out]):
    ...

class TrashCan(Generic[T_in]):
    ...
```

Is it too late to change the naming convention established in PEP 484?