Chapter 14. Inheritance: For Better or for Worse

> *[...] we needed a better theory about inheritance entirely (and still do). For example, inheritance and instancing (which is a kind of inheritance) muddles both pragmatics (such as factoring code to save space) and semantics (used for way too many tasks such as: specialization, generalization, speciation, etc.).*
>
> —Alan Kay, “The Early History of Smalltalk”[1]

This chapter is about inheritance and subclassing. I will assume a basic understanding of these concepts, which you may know from reading *The Python Tutorial* or from experience with another mainstream object-oriented language, such as Java, C#, or C++. Here we’ll focus on four characteristics of Python:

- The `super()` function
- The pitfalls of subclassing from built-in types
- Multiple inheritance and method resolution order
- Mixin classes

Multiple inheritance is the ability of a class to have more than one base class. C++ supports it; Java and C# don’t. Many consider multiple inheritance more trouble than it’s worth. It was deliberately left out of Java after its perceived abuse in early C++ codebases.

This chapter introduces multiple inheritance for those who have never used it, and provides some guidance on how to cope with single or multiple inheritance if you must use it.

As of 2021, there is a significant backlash against overuse of inheritance in general—not only multiple inheritance—because superclasses and subclasses are tightly coupled. Tight coupling means that changes to one part of the program may have unexpected and far-reaching effects in other parts, making systems brittle and hard to understand.

However, we have to maintain existing systems designed with complex class hierarchies, or use frameworks that force us to use inheritance—

even multiple inheritance sometimes.

I will illustrate practical uses of multiple inheritance with the standard library, the Django web framework, and the Tkinter GUI toolkit.

What's New in This Chapter

There are no new Python features related to the subject of this chapter, but I heavily edited it based on feedback from technical reviewers of the second edition, especially Leonardo Rochael and Caleb Hattingh.

I wrote a new opening section focusing on the `super()` built-in function, and changed the examples in "Multiple Inheritance and Method Resolution Order" for a deeper exploration of how `super()` works to support *cooperative multiple inheritance*.

"Mixin Classes" is also new. "Multiple Inheritance in the Real World" was reorganized and covers simpler mixin examples from the standard library, before the complex Django and the complicated Tkinter hierarchies.

As the chapter title suggests, the caveats of inheritance have always been one of the main themes of this chapter. But more and more developers consider it so problematic that I've added a couple of paragraphs about avoiding inheritance to the end of "Chapter Summary" and "Further Reading".

We'll start with an overview of the mysterious `super()` function.

The super() Function

Consistent use the of the `super()` built-in function is essential for maintainable object-oriented Python programs.

When a subclass overrides a method of a superclass, the overriding method usually needs to call the corresponding method of the superclass. Here's the recommended way to do it, from an example in the *collections* module documentation, section "OrderedDict Examples and Recipes":[2]

```
class LastUpdatedOrderedDict(OrderedDict):
    """Store items in the order they were last updated"""

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

To do its job, `LastUpdatedOrderedDict` overrides `__setitem__` to:

1. Use `super().__setitem__` to call that method on the superclass, to let it insert or update the key/value pair.
2. Call `self.move_to_end` to ensure the updated `key` is in the last position.

Invoking an overridden `__init__` method is particularly important to allow superclasses to do their part in initializing the instance.

TIP

If you learned object-oriented programming in Java, you may recall that a Java constructor method automatically calls the no-argument constructor of the superclass. Python doesn't do this. You must get used to writing this pattern:

```
    def __init__(self, a, b) :
        super().__init__(a, b)
        ...  # more initialization code
```

You may have seen code that doesn't use `super()`, but instead calls the method directly on the superclass, like this:

```
class NotRecommended(OrderedDict):
    """This is a counter example!"""

    def __setitem__(self, key, value):
        OrderedDict.__setitem__(self, key, value)
        self.move_to_end(key)
```

This alternative works in this particular case, but is not recommended for two reasons. First, it hardcodes the base class. The name `OrderedDict` appears in the `class` statement and also inside `__setitem__`. If in the future someone changes the `class` statement to change the base class or add another one, they may forget to update the body of `__setitem__`, introducing a bug.

The second reason is that `super` implements logic to handle class hierarchies with multiple inheritance. We'll come back to that in "Multiple Inheritance and Method Resolution Order". To conclude this refresher about `super`, it is useful to review how we had to call it in Python 2, because the old signature with two arguments is revealing:

```
class LastUpdatedOrderedDict(OrderedDict):
    """This code works in Python 2 and Python 3"""

    def __setitem__(self, key, value):
        super(LastUpdatedOrderedDict, self).__setitem__(key, value)
        self.move_to_end(key)
```

Both arguments of `super` are now optional. The Python 3 bytecode compiler automatically provides them by inspecting the surrounding context when `super()` is invoked in a method. The arguments are:

`type`

The start of the search path for the superclass implementing the desired method. By default, it is the class that owns the method where the `super()` call appears.

`object_or_type`

The object (for instance method calls) or class (for class method calls) to be the receiver of the method call. By default, it is `self` if the `super()` call happens in an instance method.

Whether you or the compiler provides those arguments, the `super()` call returns a dynamic proxy object that finds a method (such as `__setitem__` in the example) in a superclass of the `type` parameter, and binds it to the `object_or_type`, so that we don't need to pass the receiver (`self`) explicitly when invoking the method.

In Python 3, you can still explicitly provide the first and second arguments to `super()`.[3] But they are needed only in special cases, such as skipping over part of the MRO for testing or debugging, or for working around undesired behavior in a superclass.

Now let's discuss the caveats when subclassing built-in types.

Subclassing Built-In Types Is Tricky

It was not possible to subclass built-in types such as `list` or `dict` in the earliest versions of Python. Since Python 2.2, it's possible, but there is a major caveat: the code of the built-ins (written in C) usually does not call methods overridden by user-defined classes. A good short description of the problem is in the documentation for PyPy, in the "Differences between PyPy and CPython" section, "Subclasses of built-in types":

> *Officially, CPython has no rule at all for when exactly overridden method of subclasses of built-in types get implicitly called or not. As an approximation, these methods are never called by other built-in methods of the same object. For example, an overridden `__getitem__()` in a subclass of `dict` will not be called by e.g. the built-in `get()` method.*

Example 14-1 illustrates the problem.

Example 14-1. Our `__setitem__` override is ignored by the `__init__` and `__update__` methods of the built-in `dict`

```
>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)  ❶
...
>>> dd = DoppelDict(one=1)  ❷
>>> dd
{'one': 1}
>>> dd['two'] = 2  ❸
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3)  ❹
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}
```

❶ `DoppelDict.__setitem__` duplicates values when storing (for no good reason, just to have a visible effect). It works by delegating to the superclass.

❷ The `__init__` method inherited from `dict` clearly ignored that `__setitem__` was overridden: the value of `'one'` is not duplicated.

❸ The `[]` operator calls our `__setitem__` and works as expected: `'two'` maps to the duplicated value `[2, 2]`.

❹ The `update` method from `dict` does not use our version of `__setitem__` either: the value of `'three'` was not duplicated.

This built-in behavior is a violation of a basic rule of object-oriented programming: the search for methods should always start from the class of the receiver (`self`), even when the call happens inside a method implemented in a superclass. This is what is called "late binding," which Alan Kay—of Smalltalk fame—considers a key feature of object-oriented programming: in any call of the form `x.method()`, the exact method to be called must be determined at runtime, based on the class of the receiver `x`.[4] This sad state of affairs contributes to the issues we saw in "Inconsistent Usage of __missing__ in the Standard Library".

The problem is not limited to calls within an instance—whether `self.get()` calls `self.__getitem__()`—but also happens with overridden methods of other classes that should be called by the built-in methods. Example 14-2 is adapted from the PyPy documentation.

Example 14-2. The `__getitem__` of `AnswerDict` is bypassed by `dict.update`

```
>>> class AnswerDict(dict):
...     def __getitem__(self, key):  ❶
...         return 42
...
>>> ad = AnswerDict(a='foo')  ❷
>>> ad['a']  ❸
42
>>> d = {}
>>> d.update(ad)  ❹
>>> d['a']  ❺
'foo'
>>> d
{'a': 'foo'}
```

❶ `AnswerDict.__getitem__` always returns `42`, no matter what the key.

❷ `ad` is an `AnswerDict` loaded with the key-value pair `('a', 'foo')`.

❸ `ad['a']` returns `42`, as expected.

❹ `d` is an instance of plain `dict`, which we update with `ad`.

❺ The `dict.update` method ignored our `AnswerDict.__getitem__`.

WARNING

Subclassing built-in types like `dict` or `list` or `str` directly is error-prone because the built-in methods mostly ignore user-defined overrides. Instead of subclassing the built-ins, derive your classes from the `collections` module using `UserDict`, `UserList`, and `UserString`, which are designed to be easily extended.

If you subclass `collections.UserDict` instead of `dict`, the issues exposed in Examples 14-1 and 14-2 are both fixed. See Example 14-3.

Example 14-3. `DoppelDict2` and `AnswerDict2` work as expected because they extend `UserDict` and not `dict`

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
```

```
>>> class AnswerDict2(collections.UserDict):
...     def __getitem__(self, key):
...         return 42
...
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

As an experiment to measure the extra work required to subclass a built-in, I rewrote the `StrKeyDict` class from Example 3-9 to subclass `dict` instead of `UserDict`. In order to make it pass the same suite of tests, I had to implement `__init__`, `get`, and `update` because the versions inherited from `dict` refused to cooperate with the overridden `__missing__`, `__contains__`, and `__setitem__`. The `UserDict` subclass from Example 3-9 has 16 lines, while the experimental `dict` subclass ended up with 33 lines.[5]

To be clear: this section covered an issue that applies only to method delegation within the C language code of the built-in types, and only affects classes derived directly from those types. If you subclass a base class coded in Python, such as `UserDict` or `MutableMapping`, you will not be troubled by this.[6]

Now let's focus on an issue that arises with multiple inheritance: if a class has two superclasses, how does Python decide which attribute to use when we call `super().attr`, but both superclasses have an attribute with that name?

Multiple Inheritance and Method Resolution Order

Any language implementing multiple inheritance needs to deal with potential naming conflicts when superclasses implement a method by the same name. This is called the "diamond problem," illustrated in Figure 14-1 and Example 14-4.

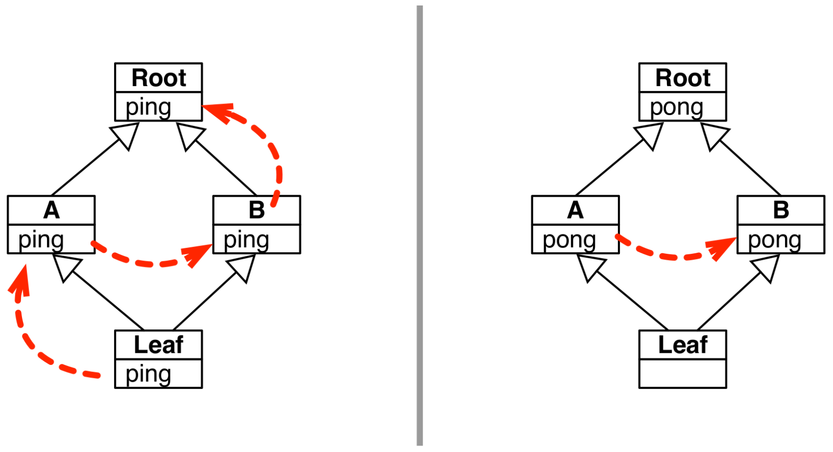

Figure 14-1. Left: Activation sequence for the `leaf1.ping()` call. Right: Activation sequence for the `leaf1.pong()` call.

Example 14-4. diamond.py: classes `Leaf`, `A`, `B`, `Root` form the graph in Figure 14-1

```
class Root:  ❶
    def ping(self):
        print(f'{self}.ping() in Root')

    def pong(self):
        print(f'{self}.pong() in Root')

    def __repr__(self):
        cls_name = type(self).__name__
        return f'<instance of {cls_name}>'


class A(Root):  ❷
    def ping(self):
        print(f'{self}.ping() in A')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in A')
        super().pong()


class B(Root):  ❸
    def ping(self):
        print(f'{self}.ping() in B')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in B')
```

```
class Leaf(A, B):  ❹
    def ping(self):
        print(f'{self}.ping() in Leaf')
        super().ping()
```

❶ `Root` provides `ping`, `pong`, and `__repr__` to make the output easier to read.

❷ The `ping` and `pong` methods in class `A` both call `super()`.

❸ Only the `ping` method in class `B` calls `super()`.

❹ Class `Leaf` implements only `ping`, and it calls `super()`.

Now let's see the effect of calling the `ping` and `pong` methods on an instance of `Leaf` (Example 14-5).

Example 14-5. Doctests for calling `ping` and `pong` on a `Leaf` object

```
>>> leaf1 = Leaf()  ❶
>>> leaf1.ping()    ❷
<instance of Leaf>.ping() in Leaf
<instance of Leaf>.ping() in A
<instance of Leaf>.ping() in B
<instance of Leaf>.ping() in Root

>>> leaf1.pong()   ❸
<instance of Leaf>.pong() in A
<instance of Leaf>.pong() in B
```

❶ `leaf1` is an instance of `Leaf`.

❷ Calling `leaf1.ping()` activates the `ping` methods in `Leaf`, `A`, `B`, and `Root`, because the `ping` methods in the first three classes call `super().ping()`.

❸ Calling `leaf1.pong()` activates `pong` in `A` via inheritance, which then calls `super.pong()`, activating `B.pong`.

The activation sequences shown in Example 14-5 and Figure 14-1 are determined by two factors:

- The method resolution order of the `Leaf` class.
- The use of `super()` in each method.

Every class has an attribute called `__mro__` holding a tuple of references to the superclasses in method resolution order, from the current class all the way to the `object` class.[7] For the `Leaf` class, this is the `__mro__`:

```
>>> Leaf.__mro__  # doctest:+NORMALIZE_WHITESPACE
    (<class 'diamond1.Leaf'>, <class 'diamond1.A'>, <class 'diamond1.B'>,
     <class 'diamond1.Root'>, <class 'object'>)
```

NOTE

Looking at Figure 14-1, you may think the MRO describes a breadth-first search, but that's just a coincidence for that particular class hierarchy. The MRO is computed by a published algorithm called C3. Its use in Python is detailed in Michele Simionato's "The Python 2.3 Method Resolution Order". It's a challenging read, but Simionato writes: "unless you make strong use of multiple inheritance and you have non-trivial hierarchies, you don't need to understand the C3 algorithm, and you can easily skip this paper."

The MRO only determines the activation order, but whether a particular method will be activated in each of the classes depends on whether each implementation calls `super()` or not.

Consider the experiment with the `pong` method. The `Leaf` class does not override it, therefore calling `leaf1.pong()` activates the implementation in the next class of `Leaf.__mro__`: the `A` class. Method `A.pong` calls `super().pong()`. The `B` class is next in the MRO, therefore `B.pong` is activated. But that method doesn't call `super().pong()`, so the activation sequence ends here.

The MRO takes into account not only the inheritance graph but also the order in which superclasses are listed in a subclass declaration. In other words, if in *diamond.py* (Example 14-4) the `Leaf` class was declared as `Leaf(B, A)`, then class `B` would appear before `A` in `Leaf.__mro__`. This would affect the activation order of the `ping` methods, and would also cause `leaf1.pong()` to activate `B.pong` via inheritance, but `A.pong` and `Root.pong` would never run, because `B.pong` doesn't call `super()`.

When a method calls `super()`, it is a *cooperative method*. Cooperative methods enable *cooperative multiple inheritance*. These terms are intentional: in order to work, multiple inheritance in Python requires the active cooperation of the methods involved. In the `B` class, `ping` cooperates, but `pong` does not.

WARNING

A noncooperative method can be the cause of subtle bugs. Many coders reading Example 14-4 may expect that when method `A.pong` calls `super.pong()`, that will ultimately activate `Root.pong`. But if `B.pong` is activated before, it drops the ball. That's why it is recommended that every method `m` of a nonroot class should call `super().m()`.

Cooperative methods must have compatible signatures, because you never know whether `A.ping` will be called before or after `B.ping`. The activation sequence depends on the order of `A` and `B` in the declaration of each subclass that inherits from both.

Python is a dynamic language, so the interaction of `super()` with the MRO is also dynamic. Example 14-6 shows a surprising result of this dynamic behavior.

Example 14-6. diamond2.py: classes to demonstrate the dynamic nature of `super()`

```
from diamond import A  ❶

class U():  ❷
    def ping(self):
        print(f'{self}.ping() in U')
        super().ping()  ❸

class LeafUA(U, A):  ❹
    def ping(self):
        print(f'{self}.ping() in LeafUA')
        super().ping()
```

❶ Class `A` comes from *diamond.py* (Example 14-4).

❷ Class `U` is unrelated to `A` or `Root` from the `diamond` module.

❸ What does `super().ping()` do? Answer: it depends. Read on.

❹ `LeafUA` subclasses `U` and `A` in this order.

If you create an instance of `U` and try to call `ping`, you get an error:

```
>>> u = U()
>>> u.ping()
Traceback (most recent call last):
  ...
AttributeError: 'super' object has no attribute 'ping'
```

The `'super' object` returned by `super()` has no attribute `'ping'` because the MRO of `U` has two classes: `U` and `object`, and the latter has no attribute named `'ping'`.

However, the `U.ping` method is not completely hopeless. Check this out:

```
>>> leaf2 = LeafUA()
>>> leaf2.ping()
<instance of LeafUA>.ping() in LeafUA
<instance of LeafUA>.ping() in U
<instance of LeafUA>.ping() in A
<instance of LeafUA>.ping() in Root
>>> LeafUA.__mro__  # doctest:+NORMALIZE_WHITESPACE
(<class 'diamond2.LeafUA'>, <class 'diamond2.U'>,
 <class 'diamond.A'>, <class 'diamond.Root'>, <class 'object'>)
```

The `super().ping()` call in `LeafUA` activates `U.ping`, which cooperates by calling `super().ping()` too, activating `A.ping`, and eventually `Root.ping`.

Note the base classes of `LeafUA` are `(U, A)` in that order. If instead the bases were `(A, U)`, then `leaf2.ping()` would never reach `U.ping`, because the `super().ping()` in `A.ping` would activate `Root.ping`, and that method does not call `super()`.

In a real program, a class like `U` could be a *mixin class*: a class intended to be used together with other classes in multiple inheritance, to provide additional functionality. We'll study that shortly, in "Mixin Classes".

To wrap up this discussion of the MRO, Figure 14-2 illustrates part of the complex multiple inheritance graph of the Tkinter GUI toolkit from the Python standard library.

Figure 14-2. Left: UML diagram of the Tkinter `Text` widget class and superclasses. Right: The long and winding path of `Text.__mro__` is drawn with dashed arrows.

To study the picture, start at the `Text` class at the bottom. The `Text` class implements a full-featured, multiline editable text widget. It provides rich functionality in itself, but also inherits many methods from other classes. The lefthand side shows a plain UML class diagram. On the right, it's decorated with arrows showing the MRO, as listed in Example 14-7 with the help of a `print_mro` convenience function.

Example 14-7. MRO of `tkinter.Text`

```
>>> def print_mro(cls):
...     print(', '.join(c.__name__ for c in cls.__mro__))
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

Now let's talk about mixins.

Mixin Classes

A mixin class is designed to be subclassed together with at least one other class in a multiple inheritance arrangement. A mixin is not supposed to be the only base class of a concrete class, because it does not provide all the functionality for a concrete object, but only adds or customizes the behavior of child or sibling classes.

NOTE

Mixin classes are a convention with no explicit language support in Python and C++. Ruby allows the explicit definition and use of modules that work as mixins—collections of methods that may be included to add functionality to a class. C#, PHP, and Rust implement traits, which are also an explicit form of mixin.

Let's see a simple but handy example of a mixin class.

Case-Insensitive Mappings

Example 14-8 shows `UpperCaseMixin`, a class designed to provide case-insensitive access to mappings with string keys, by uppercasing those keys when they are added or looked up.

Example 14-8. uppermixin.py: `UpperCaseMixin` supports case-insensitive mappings

```
import collections

def _upper(key):  ❶
    try:
        return key.upper()
    except AttributeError:
        return key

class UpperCaseMixin:  ❷
    def __setitem__(self, key, item):
        super().__setitem__(_upper(key), item)

    def __getitem__(self, key):
        return super().__getitem__(_upper(key))

    def get(self, key, default=None):
        return super().get(_upper(key), default)

    def __contains__(self, key):
        return super().__contains__(_upper(key))
```

❶ This helper function takes a `key` of any type, and tries to return `key.upper()`; if that fails, it returns the `key` unchanged.

❷ The mixin implements four essential methods of mappings, always calling `super()`, with the `key` uppercased, if possible.

Since every method ot `UpperCaseMixin` calls `super()`, this mixin depends on a sibling class that implements or inherits methods with the same signature. To make its contribution, a mixin usually needs to appear before other classes in the MRO of a subclass that uses it. In practice, that means mixins must appear first in the tuple of base classes in a class declaration. Example 14-9 shows two examples.

Example 14-9. uppermixin.py: two classes that use `UpperCaseMixin`

```
class UpperDict(UpperCaseMixin, collections.UserDict):  ❶
    pass

class UpperCounter(UpperCaseMixin, collections.Counter):  ❷
    """Specialized 'Counter' that uppercases string keys"""
❸
```

❶ `UpperDict` needs no implementation of its own, but `UpperCaseMixin` must be the first base class, otherwise the methods from `UserDict` would be called instead.

❷ `UpperCaseMixin` also works with `Counter`.

❸ Instead of `pass`, it's better to provide a docstring to satisfy the need for a body in the `class` statement syntax.

Here are some doctests from *uppermixin.py*, for `UpperDict`:

```
    >>> d = UpperDict([('a', 'letter A'), (2, 'digit two')])
    >>> list(d.keys())
    ['A', 2]
    >>> d['b'] = 'letter B'
    >>> 'b' in d
    True
    >>> d['a'], d.get('B')
    ('letter A', 'letter B')
    >>> list(d.keys())
    ['A', 2, 'B']
```

And a quick demonstration of `UpperCounter`:

```
>>> c = UpperCounter('BaNanA')
>>> c.most_common()
[('A', 3), ('N', 2), ('B', 1)]
```

`UpperDict` and `UpperCounter` seem almost magical, but I had to carefully study the code of `UserDict` and `Counter` to make `UpperCaseMixin` work with them.

For example, my first version of `UpperCaseMixin` did not provide the `get` method. That version worked with `UserDict` but not with `Counter`. The `UserDict` class inherits `get` from `collections.abc.Mapping`, and that `get` calls `__getitem__`, which I implemented. But keys were not uppercased when an `UpperCounter` was loaded upon `__init__`. That happened because `Counter.__init__` uses `Counter.update`, which in turn relies on the `get` method inherited from `dict`. However, the `get` method in the `dict` class does not call `__getitem__`. This is the heart of the issue discussed in "Inconsistent Usage of __missing__ in the Standard Library". It is also a stark reminder of the brittle and puzzling nature of programs leveraging inheritance, even at a small scale.

The next section covers several examples of multiple inheritance, often featuring mixin classes.

Multiple Inheritance in the Real World

In the *Design Patterns* book,[8] almost all the code is in C++, but the only example of multiple inheritance is the Adapter pattern. In Python, multiple inheritance is not the norm either, but there are important examples that I will comment on in this section.

ABCs Are Mixins Too

In the Python standard library, the most visible use of multiple inheritance is the `collections.abc` package. That is not controversial: after all, even Java supports multiple inheritance of interfaces, and ABCs are interface declarations that may optionally provide concrete method implementations.[9]

Python's official documentation of `collections.abc` uses the term *mixin method* for the concrete methods implemented in many of the collection ABCs. The ABCs that provide mixin methods play two roles: they are interface definitions and also mixin classes. For example, the implementation of `collections.UserDict` relies on several of the mixin methods provided by `collections.abc.MutableMapping`.

ThreadingMixIn and ForkingMixIn

The *http.server* package provides `HTTPServer` and `ThreadingHTTPServer` classes. The latter was added in Python 3.7. Its documentation says:

> *class* `http.server.ThreadingHTTPServer` *(server_address, RequestHandlerClass)*
>
> This class is identical to `HTTPServer` but uses threads to handle requests by using the `ThreadingMixIn`. This is useful to handle web browsers pre-opening sockets, on which `HTTPServer` would wait indefinitely.

This is the complete source code for the `ThreadingHTTPServer` class in Python 3.10:

```
class ThreadingHTTPServer(socketserver.ThreadingMixIn, HTTPServer):
    daemon_threads = True
```

The source code of `socketserver.ThreadingMixIn` has 38 lines, including comments and docstrings. Example 14-10 shows a summary of its implementation.

Example 14-10. Part of *Lib/socketserver.py* in Python 3.10

```
class ThreadingMixIn:
    """Mixin class to handle each request in a new thread."""

    # 8 lines omitted in book listing

    def process_request_thread(self, request, client_address):  ❶
```

```
        ... # 6 lines omitted in book listing

    def process_request(self, request, client_address):  ❷
        ... # 8 lines omitted in book listing

    def server_close(self):  ❸
        super().server_close()
        self._threads.join()
```

❶ `process_request_thread` does not call `super()` because it is a new method, not an override. Its implementation calls three instance methods that `HTTPServer` provides or inherits.

❷ This overrides the `process_request` method that `HTTPServer` inherits from `socketserver.BaseServer`, starting a thread and delegating the actual work to `process_request_thread` running in that thread. It does not call `super()`.

❸ `server_close` calls `super().server_close()` to stop taking requests, then waits for the threads started by `process_request` to finish their jobs.

The `ThreadingMixIn` appears in the `socketserver` module documentation next to `ForkingMixin`. The latter is designed to support concurrent servers based on `os.fork()`, an API for launching a child process, available in POSIX-compliant Unix-like systems.

Django Generic Views Mixins

NOTE

You don't need to know Django to follow this section. I am using a small part of the framework as a practical example of multiple inheritance, and I will try to give all the necessary background, assuming you have some experience with server-side web development in any language or framework.

In Django, a view is a callable object that takes a `request` argument—an object representing an HTTP request—and returns an object representing an HTTP response. The different responses are what interests us in this discussion. They can be as simple as a redirect response, with no content body, or as complex as a catalog page in an online store, rendered from an HTML template and listing multiple merchandise with buttons for buying, and links to detail pages.

Originally, Django provided a set of functions, called generic views, that implemented some common use cases. For example, many sites need to show search results that include information from numerous items, with the listing spanning multiple pages, and for each item a link to a page with detailed information about it. In Django, a list view and a detail view are designed to work together to solve this problem: a list view renders search results, and a detail view produces a page for each individual item.

However, the original generic views were functions, so they were not extensible. If you needed to do something similar but not exactly like a generic list view, you'd have to start from scratch.

The concept of class-based views was introduced in Django 1.3, along with a set of generic view classes organized as base classes, mixins, and ready-to-use concrete classes. In Django 3.2, the base classes and mixins are in the `base` module of the `django.views.generic` package, pictured in Figure 14-3. At the top of the diagram we see two classes that take care of very distinct responsibilities: `View` and `TemplateResponseMixin`.

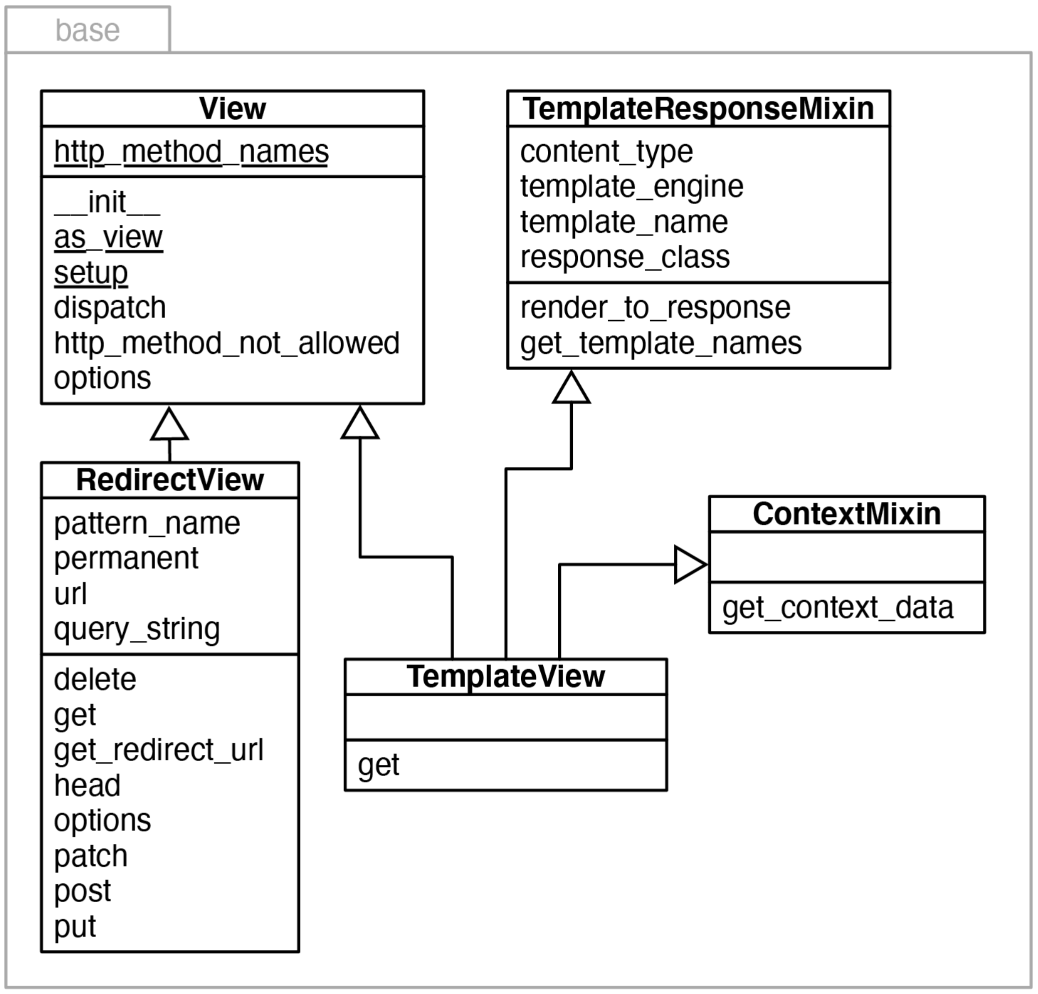

Figure 14-3. UML class diagram for the `django.views.generic.base` module.

TIP

A great resource to study these classes is the *Classy Class-Based Views* website, where you can easily navigate through them, see all methods in each class (inherited, overridden, and added methods), view diagrams, browse their documentation, and jump to their source code on GitHub.

`View` is the base class of all views (it could be an ABC), and it provides core functionality like the `dispatch` method, which delegates to "handler" methods like `get`, `head`, `post`, etc., implemented by concrete subclasses to handle the different HTTP verbs.[10] The `RedirectView` class inherits only from `View`, and you can see that it implements `get`, `head`, `post`, etc.

Concrete subclasses of `View` are supposed to implement the handler methods, so why aren't those methods part of the `View` interface? The reason: subclasses are free to implement just the handlers they want to support. A `TemplateView` is used only to display content, so it only implements `get`. If an HTTP `POST` request is sent to a `TemplateView`, the inherited `View.dispatch` method checks that there is no `post` handler, and produces an HTTP `405 Method Not Allowed` response.[11]

The `TemplateResponseMixin` provides functionality that is of interest only to views that need to use a template. A `RedirectView`, for example, has no content body, so it has no need of a template and it does not inherit from this mixin. `TemplateResponseMixin` provides behaviors to `TemplateView` and other template-rendering views, such as `ListView`, `DetailView`, etc., defined in the `django.views.generic` subpackages. Figure 14-4 depicts the `django.views.generic.list` module and part of the `base` module.

For Django users, the most important class in Figure 14-4 is `ListView`, which is an aggregate class, with no code at all (its body is just a docstring). When instantiated, a `ListView` has an `object_list` instance attribute through which the template can iterate to show the page contents, usually the result of a database query returning multiple objects. All the functionality related to generating this iterable of objects comes from the `MultipleObjectMixin`. That mixin also provides the complex pagination logic—to display part of the results in one page and links to more pages.

Suppose you want to create a view that will not render a template, but will produce a list of objects in JSON format. That's why the `BaseListView` exists. It provides an easy-to-use extension point that brings together `View` and `MultipleObjectMixin` functionality, without the overhead of the template machinery.

The Django class-based views API is a better example of multiple inheritance than Tkinter. In particular, it is easy to make sense of its mixin classes: each has a well-defined purpose, and they are all named with the `…Mixin` suffix.

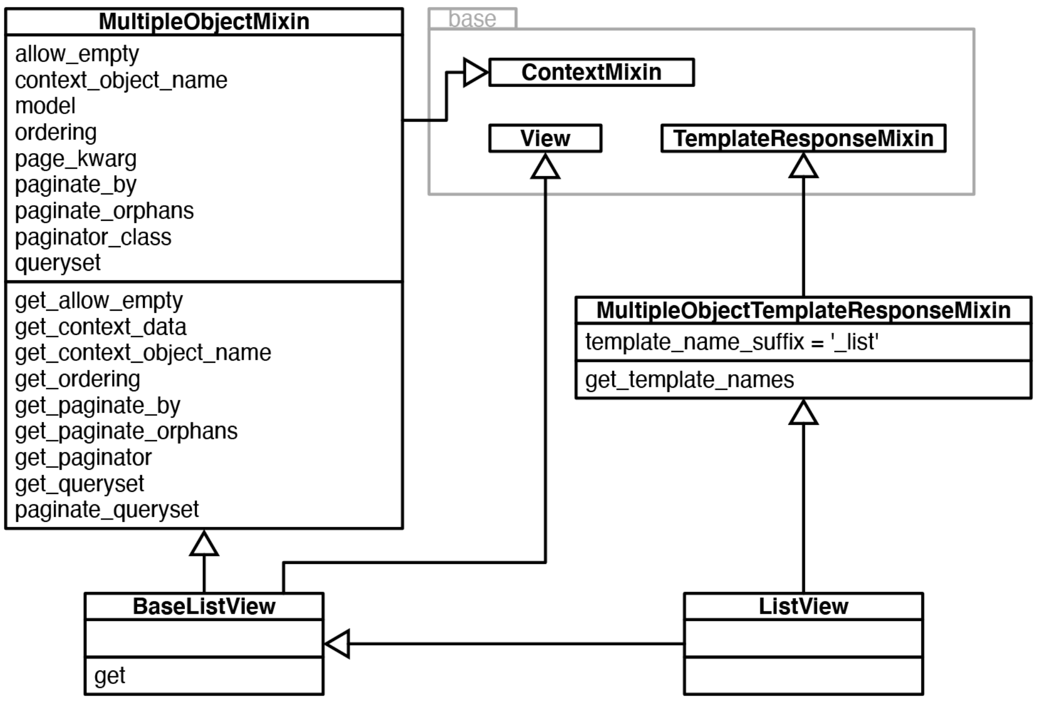

Figure 14-4. UML class diagram for the `django.views.generic.list` module. Here the three classes of the base module are collapsed (see Figure 14-3). The `ListView` class has no methods or attributes: it's an aggregate class.

Class-based views were not universally embraced by Django users. Many do use them in a limited way, as opaque boxes, but when it's necessary to create something new, a lot of Django coders continue writing monolithic view functions that take care of all those responsibilities, instead of trying to reuse the base views and mixins.

It does take some time to learn how to leverage class-based views and how to extend them to fulfill specific application needs, but I found that it was worthwhile to study them. They eliminate a lot of boilerplate code, make it easier to reuse solutions, and even improve team communication —for example, by defining standard names to templates, and to the variables passed to template contexts. Class-based views are Django views "on rails."

Multiple Inheritance in Tkinter

An extreme example of multiple inheritance in Python's standard library is the Tkinter GUI toolkit. I used part of the Tkinter widget hierarchy to illustrate the MRO in Figure 14-2. Figure 14-5 shows all the widget classes in the `tkinter` base package (there are more widgets in the `tkinter.ttk` subpackage).

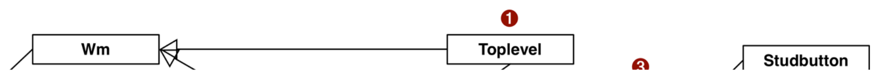

Figure 14-5. Summary UML diagram for the Tkinter GUI class hierarchy; classes tagged «mixin» are designed to provide concrete methods to other classes via multiple inheritance.

Tkinter is 25 years old as I write this. It is not an example of current best practices. But it shows how multiple inheritance was used when coders did not appreciate its drawbacks. And it will serve as a counterexample when we cover some good practices in the next section.

Consider these classes from Figure 14-5:

❶ `Toplevel` : The class of a top-level window in a Tkinter application.

❷ `Widget` : The superclass of every visible object that can be placed on a window.

❸ `Button` : A plain button widget.

❹ `Entry` : A single-line editable text field.

❺ `Text` : A multiline editable text field.

Here are the MROs of those classes, displayed by the `print_mro` function from Example 14-7:

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
```

```
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

NOTE

By current standards, the class hierarchy of Tkinter is very deep. Few parts of the Python standard library have more than three or four levels of concrete classes, and the same can be said of the Java class library. However, it is interesting to note that the some of the deepest hierarchies in the Java class library are precisely in the packages related to GUI programming: `java.awt` and `javax.swing`. Squeak, the modern, free version of Smalltalk, includes the powerful and innovative Morphic GUI toolkit, also with a deep class hierarchy. In my experience, GUI toolkits are where inheritance is most useful.

Note how these classes relate to others:

- `Toplevel` is the only graphical class that does not inherit from `Widget`, because it is the top-level window and does not behave like a widget; for example, it cannot be attached to a window or frame. `Toplevel` inherits from `Wm`, which provides direct access functions of the host window manager, like setting the window title and configuring its borders.
- `Widget` inherits directly from `BaseWidget` and from `Pack`, `Place`, and `Grid`. These last three classes are geometry managers: they are responsible for arranging widgets inside a window or frame. Each encapsulates a different layout strategy and widget placement API.
- `Button`, like most widgets, descends only from `Widget`, but indirectly from `Misc`, which provides dozens of methods to every widget.
- `Entry` subclasses `Widget` and `XView`, which support horizontal scrolling.
- `Text` subclasses from `Widget`, `XView`, and `YView` for vertical scrolling.

We’ll now discuss some good practices of multiple inheritance and see whether Tkinter goes along with them.

Coping with Inheritance

What Alan Kay wrote in the epigraph remains true: there's still no general theory about inheritance that can guide practicing programmers. What we have are rules of thumb, design patterns, "best practices," clever acronyms, taboos, etc. Some of these provide useful guidelines, but none of them are universally accepted or always applicable.

It's easy to create incomprehensible and brittle designs using inheritance, even without multiple inheritance. Because we don't have a comprehensive theory, here are a few tips to avoid spaghetti class graphs.

Favor Object Composition over Class Inheritance

The title of this subsection is the second principle of object-oriented design from the *Design Patterns* book,[12] and is the best advice I can offer here. Once you get comfortable with inheritance, it's too easy to overuse it. Placing objects in a neat hierarchy appeals to our sense of order; programmers do it just for fun.

Favoring composition leads to more flexible designs. For example, in the case of the `tkinter.Widget` class, instead of inheriting the methods from all geometry managers, widget instances could hold a reference to a geometry manager, and invoke its methods. After all, a `Widget` should not "be" a geometry manager, but could use the services of one via delegation. Then you could add a new geometry manager without touching the widget class hierarchy and without worrying about name clashes. Even with single inheritance, this principle enhances flexibility, because subclassing is a form of tight coupling, and tall inheritance trees tend to be brittle.

Composition and delegation can replace the use of mixins to make behaviors available to different classes, but cannot replace the use of interface inheritance to define a hierarchy of types.

Understand Why Inheritance Is Used in Each Case

When dealing with multiple inheritance, it's useful to keep straight the reasons why subclassing is done in each particular case. The main reasons are:

- Inheritance of interface creates a subtype, implying an “is-a” relationship. This is best done with ABCs.
- Inheritance of implementation avoids code duplication by reuse. Mixins can help with this.

In practice, both uses are often simultaneous, but whenever you can make the intent clear, do it. Inheritance for code reuse is an implementation detail, and it can often be replaced by composition and delegation. On the other hand, interface inheritance is the backbone of a framework. Interface inheritance should use only ABCs as base classes, if possible.

Make Interfaces Explicit with ABCs

In modern Python, if a class is intended to define an interface, it should be an explicit ABC or a `typing.Protocol` subclass. An ABC should subclass only `abc.ABC` or other ABCs. Multiple inheritance of ABCs is not problematic.

Use Explicit Mixins for Code Reuse

If a class is designed to provide method implementations for reuse by multiple unrelated subclasses, without implying an “is-a” relationship, it should be an explicit *mixin class*. Conceptually, a mixin does not define a new type; it merely bundles methods for reuse. A mixin should never be instantiated, and concrete classes should not inherit only from a mixin. Each mixin should provide a single specific behavior, implementing few and very closely related methods. Mixins should avoid keeping any internal state; i.e., a mixin class should not have instance attributes.

There is no formal way in Python to state that a class is a mixin, so it is highly recommended that they are named with a `Mixin` suffix.

Provide Aggregate Classes to Users

> *A class that is constructed primarily by inheriting from mixins and does not add its own structure or behavior is called an aggregate class.*
>
> —Booch et al.[13]

If some combination of ABCs or mixins is particularly useful to client code, provide a class that brings them together in a sensible way.

For example, here is the complete source code for the Django `ListView` class on the bottom right of Figure 14-4:

```
class ListView(MultipleObjectTemplateResponseMixin, BaseListView):
    """
    Render some list of objects, set by `self.model` or `self.queryset`.
    `self.queryset` can actually be any iterable of items, not just a query
    """
```

The body of `ListView` is empty, but the class provides a useful service: it brings together a mixin and a base class that should be used together.

Another example is `tkinter.Widget`, which has four base classes and no methods or attributes of its own—just a docstring. Thanks to the `Widget` aggregate class, we can create new a widget with the required mixins, without having to figure out in which order they should be declared to work as intended.

Note that aggregate classes don't have to be completely empty, but they often are.

Subclass Only Classes Designed for Subclassing

In one comment about this chapter, technical reviewer Leonardo Rochael suggested the following warning.

WARNING

Subclassing any complex class and overriding its methods is error-prone because the superclass methods may ignore the subclass overrides in unexpected ways. As much as possible, avoid overriding methods, or at least restrain yourself to subclassing classes which are designed to be easily extended, and only in the ways in which they were designed to be extended.

That's great advice, but how do we know whether or how a class was designed to be extended?

The first answer is documentation (sometimes in the form of docstrings or even comments in code). For example, Python's `socketserver` package is described as "a framework for network servers." Its `BaseServer` class is designed for subclassing, as the name suggests. More importantly, the documentation and the docstring in the source code of the class explicitly note which of its methods are intended to be overridden by subclasses.

In Python ≥ 3.8, a new way of making those design constraints explicit is provided by PEP 591—Adding a final qualifier to typing. The PEP introduces a `@final` decorator that can be applied to classes or individual methods, so that IDEs or type checkers can report misguided attempts to subclass those classes or override those methods.[14]

Avoid Subclassing from Concrete Classes

Subclassing concrete classes is more dangerous than subclassing ABCs and mixins, because instances of concrete classes usually have internal state that can easily be corrupted when you override methods that depend on that state. Even if your methods cooperate by calling `super()`, and the internal state is held in private attributes using the `__x` syntax, there are still countless ways a method override can introduce bugs.

In "Waterfowl and ABCs", Alex Martelli quotes Scott Meyer's *More Effective C++*, which says: "all non-leaf classes should be abstract." In other words, Meyer recommends that only abstract classes should be subclassed.

If you must use subclassing for code reuse, then the code intended for reuse should be in mixin methods of ABCs or in explicitly named mixin classes.

We will now analyze Tkinter from the point of view of these recommendations.

Tkinter: The Good, the Bad, and the Ugly

Most advice in the previous section is not followed by Tkinter, with the notable exception of "Provide Aggregate Classes to Users". Even then, it's not a great example, because composition would probably work better for integrating the geometry managers into `Widget`, as discussed in "Favor Object Composition over Class Inheritance".

Keep in mind that Tkinter has been part of the standard library since Python 1.1 was released in 1994. Tkinter is a layer on top of the excellent Tk GUI toolkit of the Tcl language. The Tcl/Tk combo is not originally object-oriented, so the Tk API is basically a vast catalog of functions. However, the toolkit is object-oriented in its design, if not in its original Tcl implementation.

The docstring of `tkinter.Widget` starts with the words "Internal class." This suggests that `Widget` should probably be an ABC. Although `Widget` has no methods of its own, it does define an interface. Its message is: "You can count on every Tkinter widget providing basic widget methods (`__init__`, `destroy`, and dozens of Tk API functions), in addition to the methods of all three geometry managers." We can agree that this is not a great interface definition (it's just too broad), but it is an interface, and `Widget` "defines" it as the union of the interfaces of its superclasses.

The `Tk` class, which encapsulates the GUI application logic, inherits from `Wm` and `Misc`, neither of which are abstract or mixin (`Wm` is not a proper mixin because `TopLevel` subclasses only from it). The name of the `Misc` class is—by itself—a very strong *code smell*. `Misc` has more than 100 methods, and all widgets inherit from it. Why is it necessary that every single widget has methods for clipboard handling, text selection, timer management, and the like? You can't really paste into a button or select text from a scrollbar. `Misc` should be split into several specialized mixin classes, and not all widgets should inherit from every one of those mixins.

To be fair, as a Tkinter user, you don't need to know or use multiple inheritance at all. It's an implementation detail hidden behind the widget classes that you will instantiate or subclass in your own code. But you will suffer the consequences of excessive multiple inheritance when you type `dir(tkinter.Button)` and try to find the method you need among the 214 attributes listed. And you'll need to face the complexity if you decide to implement a new Tk widget.

TIP

Despite the problems, Tkinter is stable, flexible, and provides a modern look-and-feel if you use the `tkinter.ttk` package and its themed widgets. Also, some of the original widgets, like `Canvas` and `Text`, are incredibly powerful. You can turn a `Canvas` object into a simple drag-and-drop drawing application in a matter of hours. Tkinter and Tcl/Tk are definitely worth a look if you are interested in GUI programming.

This concludes our tour through the labyrinth of inheritance.

Chapter Summary

This chapter started with a review of the `super()` function in the context of single inheritance. We then discussed the problem with subclassing built-in types: their native methods implemented in C do not call overridden methods in subclasses, except in very few special cases. That's why, when we need a custom `list`, `dict`, or `str` type, it's easier to subclass `UserList`, `UserDict`, or `UserString`—all defined in the `collections` module, which actually wrap the corresponding built-in types and delegate operations to them—three examples of favoring composition over inheritance in the standard library. If the desired behavior is very different from what the built-ins offer, it may be easier to subclass the appropriate ABC from `collections.abc` and write your own implementation.

The rest of the chapter was devoted to the double-edged sword of multiple inheritance. First we saw how the method resolution order, encoded in the `__mro__` class attribute, addresses the problem of potential naming conflicts in inherited methods. We also saw how the `super()` built-in behaves, sometimes unexpectedly, in hierarchies with multiple inheritance. The behavior of `super()` is designed to support mixin classes, which we then studied through the simple example of the `UpperCaseMixin` for case-insensitive mappings.

We saw how multiple inhertance and mixin methods are used in Python's ABCs, as well as in the `socketserver` threading and forking mixins. More complex uses of multiple inheritance were exemplified by Django's class-based views and the Tkinter GUI toolkit. Although Tkinter is not an example of modern best practices, it is an example of overly complex class hierarchies we may find in legacy systems.

To close the chapter, I presented seven recommendations to cope with inheritance, and applied some of that advice in a commentary of the Tkinter class hierarchy.

Rejecting inheritance—even single inheritance—is a modern trend. One of the most successful languages created in the 21st century is Go. It doesn't have a construct called "class," but you can build types that are structs of encapsulated fields and you can attach methods to those structs. Go allows the definition of interfaces that are checked by the compiler using structural typing, a.k.a. *static duck typing*—very similar to what we now have with protocol types since Python 3.8. Go has special syntax for building types and interfaces by composition, but it does not support inheritance—not even among interfaces.

So perhaps the best advice about inheritance is: avoid it if you can. But often, we don't have a choice: the frameworks we use impose their own design choices.

Further Reading

> *When it comes to reading clarity, properly-done composition is superior to inheritance. Since code is much more often read than written, avoid subclassing in general, but especially don't mix the various types of inheritance, and don't use subclassing for code sharing.*
>
> —Hynek Schlawack, Subclassing in Python Redux

During the final review of this book, technical reviewer Jürgen Gmach recommended Hynek Schlawack's post "Subclassing in Python Redux"—the source of the preceding quote. Schlawack is the author of the popular *attrs* package, and was a core contributor to the Twisted asynchronous programming framework, a project started by Glyph Lefkowitz in 2002. Over time, the core team realized they had overused subclassing in their design, according to Schlawack. His post is long, and cites other important posts and talks. Highly recommended.

In that same conclusion, Hynek Schlawack wrote: “Don’t forget that more often than not, a function is all you need.” I agree, and that is precisely why *Fluent Python* covers functions in depth before classes and inheritance. My goal was to show how much you can accomplish with functions leveraging existing classes from the standard library, before creating your own classes.

Subclassing built-ins, the `super` function, and advanced features like descriptors and metaclasses are all introduced in Guido van Rossum’s paper “Unifying types and classes in Python 2.2”. Nothing really important has changed in these features since then. Python 2.2 was an amazing feat of language evolution, adding several powerful new features in a coherent whole, without breaking backward compatibility. The new features were 100% opt-in. To use them, we just had to explicitly subclass `object`—directly or indirectly—to create a so-called “new style class.” In Python 3, every class subclasses `object`.

The *Python Cookbook*, 3rd ed. by David Beazley and Brian K. Jones (O’Reilly) has several recipes showing the use of `super()` and mixin classes. You can start from the illuminating section “8.7. Calling a Method on a Parent Class”, and follow the internal references from there.

Raymond Hettinger’s post “Python’s super() considered super!” explains the workings of `super` and multiple inheritance in Python from a positive perspective. It was written in response to “Python’s Super is nifty, but you can’t use it (Previously: Python’s Super Considered Harmful)” by James Knight. Martijn Pieters’ response to “How to use super() with one argument?” includes a concise and deep explanation of `super`, including its relationship with descriptors, a concept we’ll only study in Chapter 23. That’s the nature of `super`. It is simple to use in basic use cases, but is a powerful and complex tool that touches some of Python’s most advanced dynamic features, rarely found in other languages.

Despite the titles of those posts, the problem is not really the `super` built-in—which in Python 3 is not as ugly as it was in Python 2. The real issue is multiple inheritance, which is inherently complicated and tricky. Michele Simionato goes beyond criticizing and actually offers a solution in his “Setting Multiple Inheritance Straight”: he implements traits, an explict form of mixins that originated in the Self language. Simionato has a long series of blog posts about multiple inheritance in Python, including “The wonders of cooperative inheritance, or using super in Python 3”; “Mixins considered harmful," part 1 and part 2; and "Things to Know About Python Super," part 1, part 2, and part 3. The oldest posts use the Python 2 `super` syntax, but are still relevant.

I read the first edition of Grady Booch et al., *Object-Oriented Analysis and Design*, 3rd ed., and highly recommend it as a general primer on object-oriented thinking, independent of programming language. It is a rare book that covers multiple inheritance without prejudice.

Now more than ever it's fashionable to avoid inheritance, so here are two references about how to do that. Brandon Rhodes wrote "The Composition Over Inheritance Principle", part of his excellent *Python Design Patterns* guide. Augie Fackler and Nathaniel Manista presented "The End Of Object Inheritance & The Beginning Of A New Modularity" at PyCon 2013. Fackler and Manista talk about organizing systems around interfaces and functions that handle objects implementing those interfaces, avoiding the tight coupling and failure modes of classes and inheritance. That reminds me a lot of the Go way, but they advocate it for Python.

SOAPBOX

Think about the Classes You Really Need

[We] started to push on the inheritance idea as a way to let novices build on frameworks that could only be designed by experts.

—Alan Kay, “The Early History of Smalltalk”[15]

The vast majority of programmers write applications, not frameworks. Even those who do write frameworks are likely to spend a lot (if not most) of their time writing applications. When we write applications, we normally don’t need to code class hierarchies. At most, we write classes that subclass from ABCs or other classes provided by the framework. As application developers, it’s very rare that we need to write a class that will act as the superclass of another. The classes we code are almost always leaf classes (i.e., leaves of the inheritance tree).

If, while working as an application developer, you find yourself building multilevel class hierarchies, it’s likely that one or more of the following applies:

- You are reinventing the wheel. Go look for a framework or library that provides components you can reuse in your application.
- You are using a badly designed framework. Go look for an alternative.
- You are overengineering. Remember the *KISS principle*.
- You became bored coding applications and decided to start a new framework. Congratulations and good luck!

It's also possible that all of the above apply to your situation: you became bored and decided to reinvent the wheel by building your own overengineered and badly designed framework, which is forcing you to code class after class to solve trivial problems. Hopefully you are having fun, or at least getting paid for it.

Misbehaving Built-Ins: Bug or Feature?

The built-in `dict`, `list`, and `str` types are essential building blocks of Python itself, so they must be fast—any performance issues in them would severely impact pretty much everything else. That's why CPython adopted the shortcuts that cause its built-in methods to misbehave by not cooperating with methods overridden by subclasses. A possible way out of this dilemma would be to offer two implementations for each of those types: one "internal," optimized for use by the interpreter, and an external, easily extensible one.

But wait, this is what we have already: `UserDict`, `UserList`, and `UserString` are not as fast as the built-ins but are easily extensible. The pragmatic approach taken by CPython means we also get to use, in our own applications, the highly optimized implementations that are hard to subclass. Which makes sense, considering that it's not so often that we need a custom mapping, list, or string, but we use `dict`, `list`, and `str` every day. We just need to be aware of the trade-offs involved.

Inheritance Across Languages

Alan Kay coined the term "object-oriented," and Smalltalk had only single inheritance, although there are forks with various forms of multiple inheritance support, including the modern Squeak and Pharo Smalltalk dialects that support traits—a language construct that fulfills the role of a mixin class, while avoiding some of the issues with multiple inheritance.

The first popular language to implement multiple inheritance was C++, and the feature was abused enough that Java—intended as a C++ replacement—was designed without support for multiple inheritance of implementation (i.e., no mixin classes). That is, until Java 8 introduced default methods that make interfaces very similar to the abstract classes used to define interfaces in C++ and in Python. After Java, probably the most widely deployed JVM language is Scala, and it implements traits.

Other languages supporting traits are the latest stable versions of PHP and Groovy, as well as Rust and Raku—the language formerly known as Perl 6.[16] So it's fair to say that traits are trendy in 2021.

Ruby offers an original take on multiple inheritance: it does not support it, but introduces mixins as a language feature. A Ruby class can include a module in its body, so the methods defined in the module become part of the class implementation. This is a "pure" form of mixin, with no inheritance involved, and it's clear that a Ruby mixin has no influence on the type of the class where it's used. This provides the benefits of mixins, while avoiding many of its usual problems.

Two new object-oriented languages that are getting a lot of attention severely limit inheritance: Go and Julia. Both are about programming "objects," and support polymorphism, but they avoid the term "class."

Go has no inheritance at all. Julia has a type hierarchy but subtypes cannot inherit structure, only behaviors, and only abstract types can be subtyped. In addition, Julia methods are implemented using multiple dispatch—a more advanced form of the mechanism we saw in "Single Dispatch Generic Functions".

---

[1] Alan Kay, "The Early History of Smalltalk," in SIGPLAN Not. 28, 3 (March 1993), 69–95. Also available online. Thanks to my friend Christiano Anderson, who shared this reference as I was writing this chapter.

[2] I only changed the docstring in the example, because the original is misleading. It says: "Store items in the order the keys were last added," but that is not what the clearly named `LastUpdatedOrderedDict` does.

[3] It is also possible to provide only the first argument, but this not useful and may soon be deprecated, with the blessing of Guido van Rossum who created

`super()` in the first place. See the discussion at "Is it time to deprecate unbound super methods?".

4 It is interesting to note that C++ has the notion of virtual and nonvirtual methods. Virtual methods are late bound, but nonvirtual methods are bound at compile time. Although every method that we can write in Python is late bound like a virtual method, built-in objects written in C seem to have nonvirtual methods by default, at least in CPython.

5 If you are curious, the experiment is in the *14-inheritance/strkeydict_dictsub.py* file in the *fluentpython/example-code-2e* repository.

6 By the way, in this regard, PyPy behaves more "correctly" than CPython, at the expense of introducing a minor incompatibility. See "Differences between PyPy and CPython" for details.

7 Classes also have a `.mro()` method, but that's an advanced feature of metaclass programming, mentioned in "Classes as Objects". The content of the `__mro__` attribute is what matters during normal usage of a class.

8 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley).

9 As previously mentioned, Java 8 allows interfaces to provide method implementations as well. The new feature is called "Default Methods" in the official Java Tutorial.

10 Django programmers know that the `as_view` class method is the most visible part of the `View` interface, but it's not relevant to us here.

11 If you are into design patterns, you'll notice that the Django dispatch mechanism is a dynamic variation of the Template Method pattern. It's dynamic because the `View` class does not force subclasses to implement all handlers, but `dispatch` checks at runtime if a concrete handler is available for the specific request.

12 The principle appears on p. 20 of the introduction to the book.

13 Grady Booch et al., *Object-Oriented Analysis and Design with Applications*, 3rd ed. (Addison-Wesley), p. 109.

14 PEP 591 also introduces a `Final` annotation for variables or attributes that should not be reassigned or overridden.

15 Alan Kay, "The Early History of Smalltalk," in SIGPLAN Not. 28, 3 (March 1993), 69–95. Also available online. Thanks to my friend Christiano Anderson, who shared this reference as I was writing this chapter.

16 My friend and technical reviewer Leonardo Rochael explains better than I could: "The continued existence, but persistent lack of arrival, of Perl 6 was draining willpower out of the evolution of Perl itself. Now Perl continues to be developed as a separate language (it's up to version 5.34 as of now) with no shadow of deprecation because of the language formerly known as Perl 6."