Chapter 31. Designing with Classes

So far in this part of the book, we've concentrated on using Python's OOP tool, the *class*. But OOP is also about *design*—that is, how to use classes to model useful objects. Toward this end, this chapter codes common OOP design patterns in Python, such as inheritance, composition, delegation, and factories. Along the way, we'll also investigate some design-focused class concepts, such as pseudoprivate attributes, multiple inheritance, and bound methods. Because multiple inheritance is dependent on the MRO search order, we'll finally explore that here too.

One note up front: some of the design terms mentioned here require more coverage than this book can provide. If this material sparks your curiosity, you may want to consider exploring a text on OOP design or design patterns as a next step. As you'll see, the good news is that Python makes many traditional design patterns almost trivial.

Python and OOP

Let's begin with a review—Python's implementation of OOP can be summarized by three ideas:

Inheritance

Inheritance is based on attribute lookup in Python (in `X.name` expressions).

Polymorphism

In `X.method`, the meaning of `method` depends on the type (class) of subject object `X`.

Encapsulation

Methods and operators implement behavior, though data hiding is a convention by default.

By now, you should have a good feel for what basic *inheritance* is all about in Python, so we'll take it as a given here. As you've learned, it's the mechanism

behind flexible code customization.

We've also talked about Python's *polymorphism* a few times already. It flows from Python's lack of type declarations (per Chapter 6, the optional, unused, and paradoxical "type hinting" doesn't qualify). Because attributes are always resolved at runtime, objects that implement the same interfaces are automatically interchangeable; clients don't need to know what sorts of objects are implementing the methods they call.

Newer here, *encapsulation* in Python means *packaging*—that is, hiding implementation details behind an object's interface. It does not mean enforced privacy, though that can be partly implemented with code, as you'll see in Chapter 39. Encapsulation is available and useful in Python nonetheless: it allows the implementation of an object's interface to be changed without impacting the users of that object.

Polymorphism Means Interfaces, Not Call Signatures

Some OOP languages also define polymorphism to mean overloading functions based on the type signatures of their *arguments*—the number passed and/or their types. Because there are no real type declarations in Python, this concept doesn't apply; as we've seen, polymorphism in Python is based on object *interfaces*, not types.

If you're pining for your C++ days, you can try to overload methods by their argument lists, like this:

```
class C:
    def meth(self, x):
        …
    def meth(self, x, y, z):
        …
```

Such code will run without error, but because the `def` simply assigns an object to a name in the class's scope, the *last* definition of the method function is the only one that will be retained. Put another way, it's just as if you say `X = 1` and then `X = 2`; at the end, `X` will be `2`. Hence, there can be only one definition of a method name. Per Chapter 29, this includes `__init__` constructors, which are only special because of when they are run.

If call-signature dispatch is truly required, you can always code type-based selections using the type-testing ideas we met in Chapters 4 and 9, or the argument-list tools introduced in Chapter 18:

```
class C:
    def meth(self, *args):
        if len(args) == 1:                  # Branch on num
            …
        elif type(arg[0]) == int:           # Branch on arg
            …
```

You normally shouldn't do this, though—it's not the "Python way." As explained in Chapter 16, you should write your code to expect only an object *interface*, not a specific object *type*. That way, it will be useful for a broader category of types and applications, both now and in the future:

```
class C:
    def meth(self, x):
        x.operation()                       # Assume x does
```

It's also generally considered better to use distinct method *names* for distinct operations rather than relying on call signatures (no matter what language you code in).

But enough review. Although Python's object model is straightforward, much of the art in OOP is in the way we combine classes to achieve a program's goals. The next section begins a tour of some of the ways larger programs use classes to their advantage.

OOP and Inheritance: "Is-a" Relationships

We've explored the mechanics of inheritance in depth already; let's turn to an example of how it can be used to model real-world relationships. From a *programmer's* point of view, inheritance is kicked off by attribute qualifications, which trigger searches for names in instances, their classes, and then any superclasses. From a *designer's* point of view, inheritance is a way to

specify set membership: a class defines a set of properties that may be inherited and customized by more specific sets (i.e., subclasses).

To illustrate, let's put that pizza-making robot we talked about at the start of this part of the book to work. Suppose we've decided to explore alternative career paths and open a pizza restaurant (not bad, as career paths go). One of the first things we'll need to do is hire employees to serve customers, prepare the food, and so on. Being engineers at heart, we've decided to build a robot to make the pizzas, but being politically and cybernetically correct, we've also decided to make our robot a full-fledged employee with a salary.

Our pizza shop team can be simulated by the four classes in Example 31-1, *employees.py*. The most general class, `Employee`, is a takeoff on Chapter 28's demo. It provides common behavior such as bumping up salaries (`giveRaise`) and printing (`__repr__`). There are two kinds of employees, and so there are two subclasses of `Employee` — `Chef` and `Server`. Both override the inherited `work` method to print more specific messages. Finally, our pizza robot is modeled by an even more specific class— `PizzaRobot` is a kind of `Chef`, which is a kind of `Employee`. In OOP terms, we call these relationships "*is-a*" links: a robot is a chef, which is an employee.

Example 31-1. employees.py

```
class Employee:
    def __init__(self, name, salary=0):
        self.name   = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary += self.salary * percent
    def work(self):
        print(self.name, 'does stuff')
    def __repr__(self):
        return (f'<{self.__class__.__name__}: '
                f'name="{self.name}", salary={self.sala

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, 'makes food')

class Server(Employee):
```

```
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, 'interfaces with customer')

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, 'makes pizza')

if __name__ == '__main__':
    pat = PizzaRobot('pat')       # Make a robot named
    print(pat)                    # Run inherited __rep
    pat.work()                    # Run type-specific c
    pat.giveRaise(0.20)           # Give pat a 20% rais
    print(pat); print()

    for klass in Employee, Chef, Server, PizzaRobot:
        object = klass(klass.__name__)
        object.work()
```

When we run the self-test code included in this module, we create a pizza-making robot named `pat`, which inherits names from three classes: `PizzaRobot`, `Chef`, and `Employee`. For instance, printing and giving a raise to `pat` runs the `Employee` class's `__repr__` and `giveRaise` methods two levels up, respectively, simply because that's where the inheritance search finds these methods:

```
$ python3 employees.py
<PizzaRobot: name="pat", salary=50,000.00>
pat makes pizza
<PizzaRobot: name="pat", salary=60,000.00>

Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

In a class hierarchy like this, you can usually make instances of any of the classes, not just the ones at the bottom. For instance, the `for` loop in this module's self-test code creates instances of all four classes; each responds

differently when asked to work because the `work` method is different in each. `pat` the robot, for example, gets `work` from the most specific (i.e., lowest) `PizzaRobot` class.

Of course, these classes just *simulate* real-world objects; `work` prints a message for the time being, but it could be expanded to do real work later (see Python's interfaces to devices such as serial ports, Arduino boards, and the Raspberry Pi if you're taking this section much too literally!).

NOTE

The super re-reminder: Notice how Example 31-1 uses explicit class calls to run superclass constructors; this is how `PizzaRobot` salaries are set. Per the sidebar "The super Alternative", these could also use the `super().__init__(…)` form explored in the next chapter. As you'll find, this call avoids having to pass `self` along in single-inheritance trees like those here, but it's much more complex in the multiple-inheritance contexts you'll meet ahead. If you're anxious to see what this looks like now, see the alternative *employees-super.py* in the examples package.

OOP and Composition: "Has-a" Relationships

The notion of composition was introduced in Chapters 26 and 28. From a *programmer's* perspective, composition involves embedding other objects in a container object and activating them to implement container methods. To a *designer*, composition is another way to represent relationships in a problem domain. But, rather than set membership, composition has to do with components—parts of a whole.

Composition also reflects the relationships between parts, called "*has-a*" relationships. Some OOP design texts refer to composition as *aggregation* or distinguish between the two terms by using aggregation to describe a weaker dependency between container and contained. In this text, a "composition" simply refers to a collection of embedded objects. The composite class generally provides an interface all its own and implements it by directing the embedded objects.

Now that we've implemented our employees, let's put them in the pizza shop and let them get busy. Our pizza shop is a composite object: it has an oven, and it has employees like servers and chefs. When a customer enters and places an order, the components of the shop spring into action—the server takes the order, the chef makes the pizza, and so on. Example 31-2—file *pizzashop.py*—simulates all the objects and relationships in this scenario.

Example 31-2. pizzashop.py

```
from employees import PizzaRobot, Server    # From Exam

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, 'orders from', server)
    def pay(self, server):
        print(self.name, 'pays for item to', server)

class Oven:
    def bake(self):
        print('oven bakes')

class PizzaShop:
    def __init__(self):
        self.server = Server('Jan')         # Embed oth
        self.chef   = PizzaRobot('Pat')     # A robot r
        self.oven   = Oven()

    def order(self, name):
        customer = Customer(name)           # Activate
        customer.order(self.server)         # Customer
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == '__main__':
    scene = PizzaShop()                     # Make the
    scene.order('Sue')                      # Simulate
    print('...')
    scene.order('Bob')                      # Simulate
```

The `PizzaShop` class is a container and controller; its constructor makes and embeds instances of the employee classes we wrote in the prior section, as well as an `Oven` class defined here. When this module's self-test code calls the `PizzaShop` 's `order` method, the embedded objects are asked to carry out their actions in turn.

Notice that we make a new `Customer` object for each order, and we pass on the embedded `Server` object to `Customer` methods; customers come and go, but the server is part of the pizza shop composite. Also notice that employees are still involved in an inheritance relationship; composition and inheritance are complementary tools.

When we run this module, our pizza shop handles two orders—one from Sue and then one from Bob (overlapping orders with the `async` coroutines of Chapter 20 is explicitly out of scope here):

```
$ python3 pizzashop.py
Sue orders from <Server: name="Jan", salary=40,000.00>
Pat makes pizza
oven bakes
Sue pays for item to <Server: name="Jan", salary=40,000
...
Bob orders from <Server: name="Jan", salary=40,000.00>
Pat makes pizza
oven bakes
Bob pays for item to <Server: name="Jan", salary=40,000
```

Again, this is mostly just a toy simulation, but the objects and interactions are representative of composites at work. As a rule of thumb, classes can represent just about any objects and relationships you can express in a sentence; just replace *nouns* with classes (e.g., `Oven`) and *verbs* with methods (e.g., `bake`), and you'll have a first cut at a design.

Stream Processors Revisited

For a composition example that may be a bit more tangible than pizza-making robots, recall the generic data-stream processor function we partially coded in the introduction to OOP in Chapter 26, repeated here for ease:

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Rather than using a simple function here, we might code this as a class that uses composition to do its work in order to provide more structure and support inheritance. Example 31-3, file *streams.py*, demonstrates one way to code the class (it also mutates one method name, `readline`, because we're actually going to run this code here).

Example 31-3. streams.py

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer

    def process(self):
        while True:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)

    def converter(self, data):
        assert False, 'converter must be defined'
```

This class defines a `converter` method that it expects subclasses to fill in; it's an example of the *abstract superclass* model we outlined in Chapter 29 (again, more on `assert` in Part VII—it simply raises an exception if its test is false). Coded this way, `reader` and `writer` objects are embedded within the class instance (*composition*), and we supply the conversion logic in a subclass rather than passing in a converter function (*inheritance*). The file in Example 31-4, *converters.py*, shows how.

Example 31-4. converters.py

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    obj = Uppercase(open('trihack.txt'), sys.stdout)
    obj.process()
```

Here, the `Uppercase` class inherits the stream-processing loop logic of `process` (and anything else that may be coded in its superclasses). It needs to define only what is unique about it—the data conversion logic. When this file is run, it makes and runs an instance that reads from the file *trihack.txt* in the current directory and writes the uppercase equivalent of that file to the `stdout` stream (which usually means the window you're working in):

```
$ cat trihack.txt        # Use "type" on Windows
hack
Hack
HACK!

$ python3 converters.py
HACK
HACK
HACK!
```

To process different sorts of streams, pass in different sorts of objects to the class construction call. Here, we use an output file instead of a stream (be sure to exit your REPL to finalize the file):

```
$ python3
>>> import converters
>>> scan = converters.Uppercase(open('trihack.txt'), op
>>> scan.process()

$ cat trihackup.txt
HACK
```

```
HACK
HACK!
```

But, as suggested earlier, we could also pass in arbitrary objects coded as classes that define the required input and output method interfaces. Here's a simple example that passes in a writer class that wraps up the text inside HTML tags—lines are read from a file, run through uppercase conversion, and then printed with HTML tags:

```
$ python3
>>> from converters import Uppercase
>>> class HTMLize:
        def write(self, line):
            print(f'<PRE>{line.rstrip()}</PRE>')

>>> Uppercase(open('trihack.txt'), HTMLize()).process()
<PRE>HACK</PRE>
<PRE>HACK</PRE>
<PRE>HACK!</PRE>
```

If you trace through this example's control flow, you'll see that we get *both* uppercase conversion (by inheritance) and HTML formatting (by composition), even though the core processing logic in the original `Processor` superclass knows nothing about either step. The processing code only cares that writers have a `write` method and that a method named `converter` is defined; it doesn't care what those methods do when they are called. Such polymorphism and encapsulation of logic are behind much of the power of classes in Python.

As is, the `Processor` superclass only provides a file-scanning loop. In more realistic work, we might extend it to support additional programming tools for its subclasses and, in the process, turn it into a full-blown application *framework*. Coding such a tool once in a superclass enables you to reuse it in all your programs. Even in this simple example, because so much is packaged and inherited with classes, all we had to code was the HTML formatting step; the rest was free.

For another example of composition at work, see exercise 9 in "Test Your Knowledge: Part VI Exercises" and its solution in Appendix B; it's similar to the pizza shop example. We've focused on inheritance in this book because

that is the main tool that the Python language itself provides for OOP. But, in practice, composition may be used as much as inheritance as a way to structure classes, especially in larger systems. As we've seen, inheritance and composition are often complementary (and sometimes alternative) techniques. Because composition is a design issue outside the scope of the Python language and this book, though, we'll defer to other resources for more on this topic.

WHY YOU WILL CARE: CLASSES AND PERSISTENCE

We've explored Python's `pickle` and `shelve` object persistence earlier in this part of the book because it works especially well with class instances. In fact, these tools are often compelling enough to motivate the use of classes in general—by pickling or shelving a class instance, we store both data and logic.

For example, besides allowing us to simulate real-world interactions, the pizza shop classes developed in this chapter could also be used as the basis of a restaurant database. Pickling instances of such classes to a file makes them persistent across Python program executions:

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.chef
<PizzaRobot: name="Pat", salary=50,000.00>
>>> import pickle
>>> pickle.dump(shop, open('shopfile.pkl', 'wb'))
```

This stores an entire composite `shop` object in a file all at once. To bring it back later in another session or program, a single step suffices as well. Objects restored this way retain both state and behavior:

```
>>> import pickle
>>> shop = pickle.load(open('shopfile.pkl', 'rb'))
>>> shop.chef
<PizzaRobot: name="Pat", salary=50,000.00>
>>> shop.order('sue')
sue orders from <Server: name="Jan", salary=40,000.00>
Pat makes pizza
oven bakes
sue pays for item to <Server: name="Jan", salary=40,000
```

This is just a prototype as is, but we might extend the shop to keep track of inventory, revenue, and so on—saving it to its file after changes would retain its updated state. See the standard-library manual and related coverage in Chapters 9, 28, and 37 for more on pickles and shelves.

OOP and Delegation: “Like-a” Relationships

Besides inheritance and composition, object-oriented programmers often speak of *delegation*, which usually implies controller objects that embed other objects to which they pass off operation requests. The controllers can take care of administrative activities, such as logging or validating accesses, adding extra steps to interface components, or monitoring active instances.

In a sense, delegation is a special form of composition, with a single embedded object managed by a *proxy* (sometimes called a *wrapper*) class that retains most or all of the embedded object’s interface. The notion of proxies sometimes applies to other mechanisms, too, such as function calls; in delegation, we’re concerned with proxies for *all* of an object’s behavior, including method calls and other operations.

This concept was introduced by example in Chapter 28, and in Python is often implemented with the `__getattr__` method hook we studied in Chapter 30. Because this operator-overloading method intercepts accesses to nonexistent attributes, a wrapper class can use `__getattr__` to route arbitrary accesses to a wrapped object. Because this method allows attribute requests to be routed generically, the wrapper class retains the interface of the wrapped object and may add additional operations of its own.

By way of review, consider the file *trace.py* in Example 31-5.

Example 31-5. trace.py

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)              # Trac
        return getattr(self.wrapped, attrname)   # Dele
```

Recall from Chapter 30 that `__getattr__` gets the attribute name as a string. This code makes use of the `getattr` built-in function to fetch an attribute from the wrapped object by name string— `getattr(X,N)` is like

`X.N`, except that `N` is an *expression* that evaluates to a string at runtime, not a variable. In fact, `getattr(X,N)` is similar to `X.__dict__[N]`, but the former also performs an inheritance search, like `X.N`, while the latter does not (see Chapters 23 and 29 for more on the `__dict__` attribute).

You can use the approach of this module's wrapper class to manage access to any object with attributes—lists, dictionaries, and even classes and instances. Here, the `Wrapper` class simply prints a trace message on each attribute access and delegates the attribute request to the embedded `wrapped` object:

```
>>> from trace import Wrapper
>>> x = Wrapper([1, 2, 3])                  # Wrap
>>> x.append(4)                             # Dele
Trace: append
>>> x.wrapped                               # Prir
[1, 2, 3, 4]

>>> x = Wrapper({'a': 1, 'b': 2})           # Wrap
>>> list(x.keys())                          # Dele
Trace: keys
['a', 'b']
```

The net effect is to augment the entire interface of the `wrapped` object with additional code in the `Wrapper` class. We can use this to log our method calls, route method calls to extra or custom logic, adapt a class to a new interface, and so on.

In the next chapter, we'll revive the notions of wrapped objects and delegated operations as one way to extend built-in types. If you are interested in the delegation design pattern, also watch for the discussions in Chapters 32 and 39 of *function decorators*, a strongly related concept designed to augment a specific function or method call rather than the entire interface of an object, as well as *class decorators*, which serve as a way to automatically add such delegation-based wrappers to all instances of a class.

NOTE

Delegation reminder: As noted in the sidebar "Delegating Built-ins—or Not", general proxies like the `Wrapper` example here cannot directly intercept and delegate calls to operator-overloading methods run by *built-in* operations. The list's `__add__`, for instance, is not caught and fails:

```
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> [1, 2, 3].__add__([4, 5])
[1, 2, 3, 4, 5]
>>> Wrapper([1, 2, 3]) + [4, 5]
TypeError: unsupported operand type(s) for +: 'Wrapper' and '
```

Explicit-name attribute fetches are routed to `__getattr__`, but built-in operations differ in ways that impact some delegation-based tools. We'll return to this issue and see it live in Chapters 38 and 39, in the context of managed attributes and decorators. For now, keep in mind that delegation proxies may need to redefine operator-overloading methods by code, tools, or superclasses if those methods are used by embedded objects and should be routed to them.

Pseudoprivate Class Attributes

Besides larger structuring goals, class designs often must address name usage too. In Chapter 28's case study, for example, we noted that methods defined within a general tool class might be modified by subclasses if exposed, and noted the trade-offs of this policy—while it supports method customization and direct calls, it's also open to accidental replacements.

In Part V, we learned that every name assigned at the top level of a module file is exported. By default, the same holds for classes—*data hiding* is a convention, and clients may fetch or change attributes in any class or instance to which they have a reference. All attributes are "public" and "virtual," in C++ terms: they're accessible everywhere and are looked up dynamically at runtime. In fact, it's even possible to change or delete a class's method at runtime, though this is rarely done in practical programs. As a scripting language, Python is about enabling, not restricting.

All that being said, Python today does support the notion of name "mangling" (i.e., expansion) to associate some names with their classes. Mangled names

are sometimes misleadingly called “private attributes,” but really this is just a way to *localize* a name to the class that created it—name mangling does not prevent access by code outside the class. This feature is mostly intended to avoid namespace *collisions* in instances, not to restrict access to names in general; mangled names are therefore better called “pseudoprivate” than “private.”

Pseudoprivate names are an advanced and entirely optional feature, and you probably won’t find them very useful until you start writing general tools or larger class hierarchies for use in multiprogrammer projects. In fact, they are not always used even when they probably *should* be—more commonly, Python programmers code internal names with a single underscore (e.g., `_X`), which is just an informal convention to let you know that a name shouldn’t generally be changed, but this means nothing to Python itself.

Because you may see this feature in other people’s code, though, you need to be somewhat aware of it, even if you don’t use it yourself. And once you learn its advantages and contexts of use, you may find this feature to be more useful in your own code than some programmers realize.

Name Mangling Overview

Here’s how name mangling works: within a `class` statement only, any names that *start* with two underscores but do not *end* with two underscores are automatically expanded to include the name of the enclosing class at their front. For instance, a name like `__X` within a class named `Hack` is changed to `_Hack__X` automatically: the original name is prefixed with a single underscore and the enclosing class’s name. Because the modified name contains the name of the enclosing class, it’s generally unique; it won’t clash with similar names created by other classes in a hierarchy.

Name mangling happens only for names that appear inside a `class` statement’s code and then only for names that begin with two leading underscores. It works for *every* name preceded with double underscores, though—both class attributes (including method names) and instance attribute names assigned to `self`. For example, in a class named `Hack`, a method named `__meth` is mangled to `_Hack__meth`, and an instance attribute reference `self.__X` is transformed to `self._Hack__X`.

Despite the mangling, as long as the class uses the double-underscore version everywhere it refers to the name, all its references will still work. Because more than one class may add attributes to an instance, though, this mangling helps avoid clashes—but we need to move on to an example to see how.

Why Use Pseudoprivate Attributes?

One of the main issues that the pseudoprivate attribute feature is meant to alleviate has to do with the way instance attributes are stored. In Python, instance attributes normally wind up in the *single* instance object at the bottom of the class tree and are shared by class-level method functions the instance is passed into. This is different from the C++ model, where each class gets its own space for data members it defines.

Within a class's method in Python, whenever a method assigns to a `self` attribute (e.g., `self.attr = value`), it changes or creates an attribute in the instance (recall that inheritance searches happen only on reference, not on assignment). Because this is true even if multiple classes in a hierarchy assign to the same attribute, collisions are possible.

For example, suppose that when a programmer codes a class, it is assumed that the class owns the attribute name `X` in the instance. In this class's methods, the name is set and later fetched:

```
class C1:
    def meth1(self): self.X = 88          # I assume X i
    def meth2(self): print(self.X)
```

Suppose further that another programmer, working in isolation, makes the same assumption in another class:

```
class C2:
    def metha(self): self.X = 99          # Me too
    def methb(self): print(self.X)
```

Both of these classes work by themselves. The problem arises if the two classes are ever mixed together in the same class tree, using the multiple inheritance we'll expand on ahead:

```
class C3(C1, C2): ...
I = C3()                                # But only 1 X
```

Now, the value that each class gets back when it says `self.X` will depend on which class assigned it *last*. Because all assignments to `self.X` refer to the same single instance, there is only one `X` attribute— `I.X` —no matter how many classes use that attribute name.

This isn't a problem if it's expected, and indeed, this is how classes normally *communicate*—the instance is shared memory. To guarantee that an attribute belongs to the class that uses it, though, prefix the name with double underscores everywhere it is used in the class, as in Example 31-6, *pseudoprivate.py*.

Example 31-6. pseudoprivate.py

```
class C1:
    def meth1(self): self.__X = 88      # Now X is mir
    def meth2(self): print(self.__X)    # Becomes _C1_
class C2:
    def metha(self): self.__X = 99      # Me too
    def methb(self): print(self.__X)    # Becomes _C2_

class C3(C1, C2): pass
I = C3()                                # Two X names

I.meth1(); I.metha()                    # Set names
print(I.__dict__)                       # Actual storc
I.meth2(); I.methb()                    # Fetch names
```

When thus prefixed, the `X` attributes will be expanded to include the names of their classes before being added to the instance. If you run a `dir` call on `I` or inspect its namespace dictionary after the attributes have been assigned, you'll see the expanded names, `_C1__X` and `_C2__X`, but not `X`. Because the expansion makes the names more unique within the instance, the classes' coders can be fairly safe in assuming that they truly own any names that they prefix with two underscores:

```
$ python3 pseudoprivate.py
{'_C1__X': 88, '_C2__X': 99}
88
99
```

This trick can avoid potential name collisions in the instance, but note that it does not amount to true privacy. If you know the name of the enclosing class, you can still access either of these attributes anywhere you have a reference to the instance by using the fully expanded name (e.g., `I._C1__X = 77`). Moreover, names could still collide if unknowing programmers use the expanded naming pattern explicitly (unlikely, but not impossible). On the other hand, this feature makes it much less likely that you will *accidentally* step on a class's names.

Pseudoprivate attributes are also useful in larger frameworks or tools, both to avoid introducing new method names that might accidentally hide definitions elsewhere in the class tree and to reduce the chance of internal methods being replaced by names defined lower in the tree. If a method is intended for use only within a class that may be mixed into other classes, the double underscore prefix virtually ensures that the method won't interfere with other names in the tree, especially in multiple-inheritance scenarios:

```
class Super:
    def method(self): …                     # A real app

class Tool:
    def __method(self): …                   # Becomes _T
    def other(self): self.__method()        # Use my int

class Sub1(Tool, Super): …
    def actions(self): self.method()        # Runs Super

class Sub2(Tool):
    def __init__(self): self.method = 99    # Doesn't br
    def method(self): …                     # Ditto
```

We met multiple inheritance briefly in Chapter 26 and will explore it in more detail later in this chapter. Recall that superclasses are searched according to their left-to-right order in `class` header lines. Here, this means `Sub1` prefers `Tool` attributes to those in `Super`. Although in this example we

could force Python to pick the application class's methods first by switching the order of the superclasses listed in the `Sub1` class header, pseudoprivate attributes resolve the issue altogether. Pseudoprivate names also prevent subclasses from accidentally redefining the internal method's names, as in `Sub2`.

Again, this feature tends to be of use primarily for larger multiprogrammer projects and then only for selected names. Don't be tempted to clutter your code unnecessarily; only use this feature for names that truly need to be controlled by a single class. Although useful in some general class-based tools, for simpler programs, it's probably overkill.

For more examples that make use of the `__X` naming feature, see the *lister.py* mix-in classes introduced later in this chapter's multiple inheritance section, as well as the later class decorators mentioned in the following note.

NOTE

Private matters: If you're interested in more binding forms of privacy, you may want to review the emulation of private instance attributes coded in "Attribute Access: `__getattr__` and `__setattr__`" in Chapter 30 and watch for the broader `Private` class decorator we'll build with delegation in Chapter 39. Although it's possible to add name-access controls in Python classes, this is rarely done in practice—even for large systems that solve real-world problems. Go figure?

Method Objects: Bound or Not

Methods in general, and bound methods in particular, simplify the implementation of many design goals in Python. We met bound methods briefly while studying `__call__` in Chapter 30. The full story, which we'll flesh out here, turns out to be more general and flexible than you might expect.

In Chapter 19, we learned how functions can be processed as normal objects. Methods are a kind of object too, and can be used generically in much the same way as other objects—they can be assigned to names, passed to functions, stored in data structures, and so on—and like simple functions, modules, and classes, qualify as *first-class objects*. Because a class's methods can be accessed from an instance or a class, though, they come in two flavors:

Bound methods: when a method is referenced through an instance

Accessing a function attribute of a class by qualifying an *instance* returns a bound method object. This object automatically packages the instance with the function as a pair. When a bound method is later called, the instance is automatically passed to the function's `self` argument.

Plain functions: when a method is referenced through a class

Accessing a function attribute of a class by qualifying a *class* returns a plain function object. To call this function later, you must provide an instance object explicitly to the `self` argument—if the function expects one.

Both kinds of methods are full-fledged objects; they can be transferred around a program at will, just like strings and numbers. Bound methods simply remember an instance, but plain functions do not. This is why we've had to pass in an instance explicitly when calling superclass methods from subclass methods in previous examples (including this chapter's *employees.py* in Example 31-1); technically, such calls produce plain functions along the way.

When calling a *bound* method object, though, Python provides the instance argument for us—the instance that was used to create the bound method object. This means that bound method objects are usually interchangeable with simple function objects and makes them especially useful for interfaces originally written for functions.

NOTE

Blast from the past: Python once required all methods in a class to have an instance argument and termed methods fetched directly from a class *unbound methods*—implying that they required an instance argument when later called. Today, method functions in a class are really just plain functions; their only special quality is that they are bound to an instance when fetched through the instance. Unbound methods, however, are dead; long live plain functions!

Bound Methods in Action

To illustrate in simple terms, suppose we define the following class in the REPL of our choosing:

```
>>> class Hack:
        def doit(self, message):
            print(message)
```

Now, in normal operation, we make an instance and call its method in a single step to print the passed-in argument:

```
>>> inst = Hack()
>>> inst.doit('hello')     # Typical method calls
hello
```

Really, though, a *bound* method object is generated along the way—just before the method call's parentheses. In fact, we can fetch a bound method without actually calling it. An *`object.name`* expression evaluates to an object as all expressions do. In the following, it returns a bound method object that packages the instance (`inst`) with the method function (`Hack.doit`). The net effect is that we can assign this bound method pair to another name and then call it as though it were a simple self-less function:

```
>>> inst = Hack()
>>> meth = inst.doit       # Bound method object: insta
>>> meth('hola')           # Same effect as inst.doit('
hola
```

In fact, if you know where to look, you can see the instance/function pair that the bound method packages:

```
>>> meth
<bound method Hack.doit of <__main__.Hack object at 0x1
>>> meth.__self__
<__main__.Hack object at 0x108f2e8a0>
>>> meth.__func__
<function Hack.doit at 0x108f37ce0>
```

On the other hand, if we qualify the *class* to get to `doit` , we get back a plain function object with no associated instance. To call this type of method, we usually must pass in an instance as the leftmost argument—there isn't one in the expression otherwise, and the method in this demo expects it:

```
>>> inst = Hack()
>>> meth = Hack.doit       # Plain function: requires s
>>> meth(inst, 'ciao')     # Pass in instance (if the m
ciao
```

This time, though, the method is substantially more mundane because it's a plain function:

```
>>> meth
<function Hack.doit at 0x108f37ce0>
```

By extension, the same rules apply within a class's method if we reference `self` attributes that name functions in a class. A `self.method` expression is a bound method object because `self` *is* an instance object, though a `class.method` is a simple function that may require a `self` when run:

```
>>> class Hack2(Hack):
        def doit2(self):
            meth = self.doit        # Bound method obje
            meth('bonjour')         # Looks like a simp
            meth = Hack.doit
            meth(self, 'privit')    # Plain function: r

>>> Hack2().doit2()
bonjour
privit
```

Most of the time, you call methods immediately after fetching them with attribute qualification, so you don't notice the method objects generated along the way. But if you start writing code that calls objects generically, you need to be careful to treat nonbound methods specially—they normally require an explicit instance object to be passed in.

Implied by this model, classes can also code methods that do *not* require a `self` instance, and are called normally when fetched from their *class*; apart from inheritance, this is similar to a function in a module file:

```
>>> class Hack3:
        def doit3(message):         # A self-less metho
```

```
            print(message)

>>> Hack3.doit3('guten tag')       # No "self" is passe
guten tag
```

This falls down, though, if we try to call such a self-less method function through an *instance*: because the bound method made along the way packages and passes an instance, the call sends one too many arguments:

```
>>> x = Hack3()
>>> x.doit3('namaste')
TypeError: Hack3.doit3() takes 1 positional argument bu
```

In other words, you can code self-less functions in a class, and can call them normally through a class without an instance. To call them though an instance, too, though, you'll have to stay tuned for the next chapter's coverage of *static* and *class* methods—methods marked specially to suppress an automatic instance argument in all contexts. Also in the next chapter, you'll see that the `super` built-in binds an instance with a method, too, but is substantially more convoluted than the bound method pairs we've met here.

Finally, to demo how flexible bound methods can be, the following stores four of them in a list and calls them generically, with normal call expressions:

```
>>> class Number:
        def __init__(self, base):
            self.base = base
        def double(self):
            return self.base * 2
        def triple(self):
            return self.base * 3

>>> x, y, z = Number(2), Number(3), Number(4)
>>> x.double()
4
>>> acts = [x.double, y.double, z.double, z.triple]
>>> for act in acts:
        print(act(), end=' ')

4 6 8 12
```

In the end, calls to `act` here run methods in the class to process instances—both saved previously in bound methods.

WHY YOU WILL CARE: BOUND METHOD CALLBACKS

Because bound methods automatically pair an instance with a class's method function, you can use them anywhere a simple function is expected. One of the most common places you'll see this idea put to work is in code that registers methods to handle event callbacks run by GUI interfaces, like Python's `tkinter` standard-library module. As review, here's the simple case:

```
def handler():
    …use globals or closure scopes for state…
…
widget = Button(text='Tap', command=handler)
```

With `tkinter`, to register a handler for button click events, we usually pass a callable object that takes no arguments to the `command` keyword argument. Function names (and `lambda`s) work here, and so do class-level methods—though they must be bound methods if they expect an instance when called:

```
class MyGui:
    def handler(self):
        …use self.attr for state…
    def makewidgets(self):
        b = Button(text='Tap', command=self.handler)
```

Here, the event handler is `self.handler`—a bound method object that remembers both `self` and `MyGui.handler`. Because `self` will refer to the original instance when `handler` is later invoked on events, the method will have access to instance attributes that can retain state between events, as well as class-level methods. With simple functions, state normally must be retained in global variables or enclosing function scopes (a.k.a. closures) instead.

See also the discussion of `__call__` operator overloading in Chapter 30 for another way to make classes compatible with function-based APIs, as well as `lambda` in Chapter 19 for another tool often used in callback roles. As noted in the former of these, you don't need to wrap a bound method in a `lambda` unless extra arguments must be added to the call; because the bound method

in the preceding example *already* defers the call (there are no parentheses to trigger one!), adding a `lambda` here would otherwise be pointless.

Classes Are Objects: Generic Object Factories

Sometimes, class-based designs require objects to be created in response to conditions that can't be predicted when a program is written. The factory design pattern allows such a deferred approach. Due in large part to Python's flexibility, factories can take multiple forms, some of which don't seem special at all.

Because classes are also first-class objects (in the Chapter 19 sense), it's easy to pass them around a program, store them in data structures, and so on. You can also pass classes to functions that generate arbitrary kinds of objects; such functions are sometimes called *factories* in OOP design circles. Factories can be a major undertaking in a statically typed language such as C++ but are almost trivial to implement in Python.

For example, the call syntax we studied in Chapter 18 can call any class with any number of positional or keyword constructor arguments in one step to generate any sort of instance. Example 31-7 demos the underlying code.

Example 31-7. factory.py

```
def factory(aClass, *pargs, **kargs):        # Varargs
    return aClass(*pargs, **kargs)           # Call aCl

class Hack:
    def doit(self, message):
        print(message)

class Person:
    def __init__(self, name, job=None):
        self.name = name
        self.job  = job

object1 = factory(Hack)                      # Make a H
```

```
object2 = factory(Person, 'Sue', 'dev')      # Make a P
object3 = factory(Person, name='Bob')        # Ditto, w
```

This code's `factory` is passed a class object, along with zero or more arguments for the class's constructor. When called, it uses star syntax to collect and unpack arguments and calls the class to return an instance. Really, `factory` can invoke any callable object, including functions, classes, and methods, but we're using it for classes here.

The rest of the example simply defines two classes and generates instances of both by passing them to the `factory` function. And that's the only factory function you may ever need to write in Python; it works for any class and any constructor arguments. If you run this example live, your objects will look like this:

```
>>> from factory import *
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Sue', 'dev')
>>> object3.name, object3.job
('Bob', None)
```

By now, you should know that everything is a first-class object in Python—including classes, which are usually just compiler input in languages like C++. It's natural to pass them around this way. As mentioned at the start of this part of the book, though, only objects *derived* from classes do full OOP in Python.

Why Factories?

So, what good is the `factory` function (besides providing an excuse to illustrate first-class class objects in this book)? Unfortunately, it's difficult to show applications of this design pattern without listing much more code than we have space for here. In general, though, such a factory might allow code to be insulated from the details of dynamically configured object construction.

For instance, recall the `Processor` class presented as a composition demo earlier in Example 31-3. It accepts reader and writer objects for processing arbitrary data streams. The original abstract version of this example in Chapter 26 manually passed in instances of specialized classes like

`FileWriter` and `SocketReader` to customize the data streams being processed; later, we passed in hardcoded file, stream, and formatter objects. In a more dynamic scenario, external devices such as configuration files or GUIs might be used to configure the streams.

In such a dynamic world, we might not be able to hardcode the creation of stream interface objects in our scripts but might instead create them at runtime according to the contents of a configuration file.

Such a file might simply give the string name of a stream class to be imported from a module, plus an optional constructor call argument. Factory-style functions or code might come in handy here because they would allow us to fetch and pass in classes that are not hardcoded in our program ahead of time. Indeed, those classes might not even have existed at all when we wrote our code. Hypothetically:

```
classname = …parse from config file…
classarg  = …parse from config file…

import streamtypes                              # Customiz
aclass = getattr(streamtypes, classname)        # Fetch fr
reader = factory(aclass, classarg)              # Or aclas
processor(reader, …)
```

Here, the `getattr` built-in is again used to fetch a module attribute given a string name (it's like saying *`obj.attr`*, but *`attr`* is a string). This code snippet doesn't strictly need `factory`—it could make an instance with just `aclass(classarg)`. A separate function, though, may prove more useful when extra work is required at instance creation time, such as caching objects for reuse. However they are coded, factories are almost trivial with Python's dynamic typing and universal first-class object model.

Multiple Inheritance and the MRO

Our last design pattern is one of the most useful and will serve as a subject for more realistic examples to wrap up this chapter. As a bonus, the code we'll write here may be useful tools.

Most of our examples so far have used *single inheritance*—class trees in which each class has just one superclass. This suffices for simple hierarchies and enables customization. In our pizza shop demo of Example 31-1, for example, each worker belonged to just one category and inherited names from only one branch of the class tree. Abstractly:

```
class B: …
class A(B): …
I = A()                 # Use attributes from I, A, and
```

Many class-based designs, however, call for *combining* disparate sets of methods. As we've seen, in a `class` statement, more than one superclass can be listed in parentheses in the header line. When you do this, you leverage *multiple inheritance*—the class and its instances inherit names from *all* the listed superclasses:

```
class C: …
class B: …
class A(B, C): …
I = A()                 # Use attributes from I, A, B, C
```

In general, multiple inheritance is good for modeling objects that belong to more than one set. For instance, a person may be an engineer, a writer, a musician, and so on and inherit properties from all such sets. With multiple inheritance, objects obtain the *union* of the behavior in all their superclasses. As you'll see ahead, multiple inheritance also allows classes to function as general packages of mixable attributes known as *mix-in* classes.

Although it's a useful tool, multiple inheritance adds another dimension to attribute inheritance:

- *Single* inheritance searches only *depth first*—from instance, to class, and to each superclass from lowest to highest. This order is determined by the class from which an instance is made and the superclass listed in parentheses in each `class` statement's header (which is mirrored by class objects' `__bases__` attributes).
- *Multiple* inheritance also searches *left to right*—according to the order of classes listed in parentheses in `class` headers. This is a nested

component, pursued only after branches further to the left have reached the top of the tree and have been exhausted of their own right-branch candidates.

We call the combination of these two *DFLR*, for depth first, and then left to right. This search suffices when an attribute name shows up in just one branch of a class tree, which is a typical case. When the same name appears in multiple branches, though, DFLR alone is subpar: a lower (and hence more specialized) subclass to the right isn't able to redefine a name in one of its higher (and hence more general) superclasses reached through a branch to the left.

Because of this, multiple inheritance requires a slightly different search order known as *MRO*, for method resolution order (though it's used for all attributes, not just methods). In brief, MRO order works the same as DFLR in typical trees but proceeds *across* by tree levels before moving up in a more breadth-first fashion when multiple classes in a tree share a common superclass, forming what's called a *diamond* pattern—after the tree's square-on-its-corner shape.

This MRO search order is run for all class trees but differs from DFLR when multiple-inheritance diamonds are present. It's designed to visit a common superclass just *once*, and *after* all its subclasses. While user-defined classes don't often form diamonds in Python, the built-in `object` class automatically added above all *root* (topmost) classes makes every multiple-inheritance tree a diamond; without MRO, `object`'s defaults may hide user-defined versions.

To illustrate what diamonds are all about and how MRO search runs across before up in this one somewhat atypical case, let's turn to some code.

How Multiple Inheritance Works

We'll get more formal about MROs in the next section, but it's easy to demo live. Let's start with the simple and typical case. In the following, class `A` inherits from both `B` and `C` using multiple inheritance, and `B` and `C` are root (topmost) classes that define unique attributes. When we make an instance `I` of `A` and ask for its attributes, inheritance searches `I`, `A`, `B`, and `C`—and in that order. Hence, `attr1` is located in `B`, and `attr2` in `C`

(if you're working along in a REPL, type a blank line after each `class`, omitted here for brevity):

```
>>> class C:       attr2 = 'C2'
>>> class B:       attr1 = 'B1'
>>> class A(B, C): pass

>>> I = A()
>>> I.attr1, I.attr2
('B1', 'C2')
```

As you might already expect, `B` wins if it has the *same* name as `C` due to the left-to-right component of the search, and lower classes like `A` still beat their supers as before:

```
>>> class C:       attr2 = 'C2'
>>> class B:       attr2 = 'B2'; attr1 = 'B1'
>>> class A(B, C): attr1 = 'A1'

>>> I = A()
>>> I.attr1, I.attr2
('A1', 'B2')
```

Next, let's add a higher superclass above `B`: in the following, `A` inherits from both `B` and `C` as before, but `B` also inherits from `D`, and `C` is a root class with no supers. Per the DFLR order (depth first and then left to right), `attr2` is found in `D` by virtue of searching `I`, `A`, `B`, and `D`—and before `C` is even checked:

```
>>> class D:       attr2 = 'D2'
>>> class C:       attr2 = 'C2'
>>> class B(D):    attr1 = 'B1'
>>> class A(B, C): pass

>>> I = A()
>>> I.attr1, I.attr2
('B1', 'D2')
```

Watch what happens, though, if `B` and `C` *both* inherit from `D`, and `C` has the same attribute as `D`—per the MRO, the lower `C` class's attribute wins and effectively replaces that attribute in `D`:

```
>>> class D:       attr2 = 'D2'
>>> class C(D):    attr2 = 'C2'
>>> class B(D):    attr1 = 'B1'
>>> class A(B, C): pass

>>> I = A()
>>> I.attr1, I.attr2
('B1', 'C2')
```

This last case is what is meant by a *diamond* pattern of inheritance: multiple classes, `B` and `C`, both inherit from the same superclass. When this happens, the lower class's version of an attribute is used instead of a same-named version in a superclass—even if that superclass could be reached first by depth alone.

Importantly, this differs only in diamonds: *nondiamonds* still choose higher superclasses over lower classes with the same attribute name as in the third example in this section. The MRO allows a class to override attributes in a higher class from which it inherits; in the last example, `C` can replace names in `D` even if `D` is first per DFLR.

To summarize: in *nondiamonds*, the search proceeds all the way to the top, then backs up and starts searching to the right (DFLR); in *diamonds*, the search checks classes to the right before climbing up to a common superclass (MRO).

Strictly speaking, all four examples in this section are *implicit* diamonds because root classes inherit from the built-in `object` class automatically. We've seen that this class provides defaults for some methods, such as print displays. Because of the MRO, the `object` built-in's defaults never hide methods in user-defined classes below it in a class tree. To see how `object` is ruled out this way, let's get a bit more precise on the MRO's order.

How the MRO Works

Formally speaking, the MRO inheritance search order works as if all classes are listed per the DFLR, and then all but the rightmost duplicate of each class is removed. In more detail, it's computed as follows:

1. List all the classes from which an instance inherits using the DFLR lookup order, and include a class multiple times if it's visited more than once.
2. Scan the resulting list for duplicate classes, removing all but the last (rightmost) occurrence of duplicates in the list.

The resulting MRO sequence for a given class includes the class, its superclasses, and all higher superclasses up to and including the implicit or explicit `object` root class above the tops of the tree. It's ordered such that each class appears before its parents, and multiple parents retain the order in which they appear in the `class` header.

Because common parents in diamonds appear only at the position of their *last* visitation in the MRO, lower classes are searched first when the MRO list is used later by attribute inheritance (making it more breadth-first than depth-first in diamonds only), and each class is included and thus visited just once, no matter how many classes lead to it.

Consider the *nondiamond* class tree of Example 31-8, whose shape is sketched as "ASCII art" in its comments.

Example 31-8. mro_nondiamond.py

```
class E:       attr = 'E'     #   D     E
class D:       attr = 'D'     #   |     |
class C(E):    attr = 'C'     #   B     C
class B(D):    pass           #    \   /
class A(B, C): pass           #      A
                              #      |
X = A()                       #      X
print(X.attr)  # D
```

To compute the MRO inheritance search order through this tree, Python first enumerates all classes accessible per DFLR and then removes duplicates (remember that the built-in `object` is implicitly above all root classes):

```
DFLR => [X, A, B, D, object, C, E, object]
MRO  => [X, A, B, D, C, E, object]
```

In this tree, the net result for both DFLR and MRO ordering is the same: the output is "D" by taking `attr` from class `D` at the top left. Technically, the

MRO differs because the built-in `object` appears just once at the end after its duplicates are removed; this is irrelevant to our classes because they don't redefine an `object` default (like print strings). When a user-defined *diamond* is coded in Example 31-9, however, the MRO's difference is more striking.

Example 31-9. mro_diamond.py

```
class D:       attr = 'D'     #      D
class C(D):    attr = 'C'     #    /   \
class B(D):    pass           #   B     C
class A(B, C): pass           #    \   /
                              #      A
X = A()                       #      |
print(X.attr)  # C            #      X
```

For this tree, the common user-defined class `D` is reached *twice* by DFLR. Because the MRO keeps only the last (rightmost) `D`, the lower `C`'s definition of `attr` now wins over `D`'s. Hence, the output is now "C" instead of "D":

```
DFLR => [X, A, B, D, object, C, D, object]
MRO  => [X, A, B, C, D, object]
```

In fact, you can view the MRO of any class with its built-in `__mro__` attribute. This tuple gives the inheritance search order followed for instances of the class (after the instance itself). It's set to the result of the `mro` class method at class creation time (which can technically be customized for roles too obscure to cover here). It's also a lot to look at unless we select class names as in the following, which inspects classes in the *nondiamond* tree of Example 31-8:

```
>>> from mro_nondiamond import *
D
>>> [c.__name__ for c in A.__mro__]
['A', 'B', 'D', 'C', 'E', 'object']
>>> [c.__name__ for c in C.__mro__]
['C', 'E', 'object']
```

And here's the case for the *diamond* class tree in Example 31-9:

```
>>> from mro_diamond import *
C
>>> [c.__name__ for c in A.__mro__]
['A', 'B', 'C', 'D', 'object']
>>> [c.__name__ for c in C.__mro__]
['C', 'D', 'object']
```

The full `__mro__` is the classes used by inheritance at a given class, and `__bases__` is just supers there:

```
>>> A.__mro__
(<class 'mro_diamond.A'>, <class 'mro_diamond.B'>, <cla
<class 'mro_diamond.D'>, <class 'object'>)

>>> X.__class__.__mro__
(<class 'mro_diamond.A'>, <class 'mro_diamond.B'>, <cla
<class 'mro_diamond.D'>, <class 'object'>)

>>> A.__bases__
(<class 'mro_diamond.B'>, <class 'mro_diamond.C'>)

>>> D.__bases__
(<class 'object'>,)
```

Experiment with `__mro__` on your own for more fidelity. Especially if you're unsure how the MRO handles a given class tree, this attribute can be consulted to see how inheritance will truly search.

NOTE

Fine print: There's more to the MRO algorithm than this book can cover. Its brief sketch here is an incomplete approximation by design and won't yield accurate results for some complex class trees. You can find all the gory MRO details in Python's docs, but keeping your class trees simple will also keep them immune from the more convoluted bits of this wildly esoteric algorithm. Whether in your code or Python itself, obfuscation is rarely good engineering.

Attribute Conflict Resolution

Though a useful pattern, multiple inheritance's chief downside is that it can pose a *conflict* when the same method (or other attribute) name is defined in more than one branch of the class tree. When this occurs, the conflict is resolved either automatically by the inheritance search order or manually in your code:

Default

By default, inheritance chooses the *first* occurrence of an attribute it finds when an attribute is referenced normally (e.g., by `self.attr`). In this mode, Python chooses the first appearance located while scanning the MRO of an instance's class, from left to right.

Explicit

In some class models, you may need to *select* an attribute explicitly by referencing it through its class name (e.g., by `superclass.attr`). Your code resolves the conflict this way and overrides the search's default to select an option other than the inheritance search's default.

By default, multiple inheritance assumes that names on the *left* of a class tree should override the same names on the right, and the MRO further assumes that *lower* classes should override same-named attributes in common superclasses of diamonds. Of course, the problem with assumptions is that they assume things. In Example 31-9, what if you need some same-named attributes from `B` but *also* some others from `C`?

Luckily, there is an inheritance escape hatch. If the default search order doesn't work, or if you simply want more control over the search process, you can always force the selection of an attribute from anywhere in the tree by assigning or otherwise naming the one you want at the place where classes are mixed together.

In Example 31-9, for instance, any of the following selections are allowed, but the latter two make the choice explicit rather than relying on the implicit and subtle "magic" of MRO inheritance:

```
class A(B, C): pass              # Use the default MRO
class A(B, C): attr = B.attr     # Choose attr from the
```

```
class A(B, C): attr = C.attr      # Choose attr from the
```

Naturally, attributes picked this way can also be methods—which are normal, assignable attributes that happen to reference callable function objects. Moreover, the choice can be made in a call:

```
class A(B, C):
    def …:
        self.method(…)            # Use the default MRO

class A(B, C):
    def …:
        B.method(self, …)         # Choose method from t

class A(B, C):
    def …:
        C.method(self, …)         # Choose method from t
```

Such calls can be used to override the MRO, but also to kick calls up the tree when the class's own version must be skipped. As you'll see in the next chapter, such calls might also use `super().method()`, which selects the class following the call's host on `self`'s MRO; though simple in single inheritance, this can be stunningly implicit in multiple inheritance and doesn't provide as much control as explicit class names.

However they are coded, such manual overrides are required only when the *same name* appears in multiple superclasses, and you do not wish to use the first one inherited. For example, manual overrides allow you to unambiguously choose among a set of same names in *both* left and right branches, while inheritance would choose just names on the left. They can also be used to make choices explicit in general, though, even in nondiamonds.

Because this isn't as common an issue in typical Python code as it may sound, we'll defer details on this topic until we study the `super` built-in in the next chapter and revisit this as a "gotcha" at the end of that chapter. First, though, the next section demonstrates a practical use case for the multiple-inheritance design pattern.

NOTE

The inheritance finale: Despite the MRO's complexity, there are still a few inheritance hurdles left to clear. Its complete algorithm is more complex than the model sketched here, incorporating special cases for metaclasses, descriptors, and built-ins. In Chapter 32, we'll expand inheritance for the metaclass tree and apply its MRO in the `super` built-in, and in Chapter 38, we'll study its special case for built-in operations, but we won't be able to formalize inheritance in full until Chapter 40 after we've studied all these tools in more depth. The good news is that the remaining hurdles almost never trip up application programs; for most coders, the "finale" is a fully optional read.

Example: "Mix-in" Attribute Listers

Perhaps the most common way multiple inheritance is used is to "mix in" general-purpose methods from superclasses. Such superclasses are usually called *mix-in classes*—they provide methods you add to application classes by inheritance. In a sense, mix-in classes are similar to modules: they provide packages of methods for use in their client subclasses. Unlike simple functions in modules, though, methods in mix-in classes also can participate in inheritance hierarchies and have access to the `self` instance for using state information and other methods in their trees.

For example, we've seen that Python's default way to print a class instance object isn't incredibly useful:

```
>>> class Hack:
        def __init__(self, what):              # No __
            self.data1 = what

>>> X = Hack('code')
>>> print(X)                                   # Defau
<__main__.Hack object at 0x10ba464e0>
```

As you learned in both Chapter 28's case study and Chapter 30's operator-overloading coverage, classes can provide a `__str__` or `__repr__` method to implement custom displays. But rather than coding one of these in each class you wish to print, why not code it once in a general-purpose tool class and inherit it in all your other classes?

That's what mix-ins are for. Defining a display method in a mix-in superclass once enables us to reuse it anywhere we want to see a custom display format—even in classes that may already have another superclass. We've already explored tools that do related work:

- Chapter 28's `AttrDisplay` class, of Example 28-13, formatted instance attributes in a generic `__repr__` method, but it did not climb class trees and was utilized in single-inheritance mode only.
- Chapter 29's *classtree.py* module, of Example 29-9, defined functions for climbing and sketching class trees, but it did not display object attributes along the way and was not architected as an inheritable class.

Here, we're going to revisit these examples' techniques and expand upon them to code a set of three mix-in classes that serve as generic display tools for listing instance attributes, inherited attributes, and attributes on all objects in a class tree, respectively. We'll also use our tools in multiple-inheritance mode and deploy coding techniques that make classes better suited to use as generic tools.

Unlike Chapter 28, we'll code this with a `__str__` instead of a `__repr__`. This is partially a style issue and limits their role to `print` and `str`, but the displays we'll be developing are meant to be user-friendly, not imitative of code. This policy also leaves client classes the option of coding an alternative lower-level display for interactive echoes and nested appearances with a `__repr__` and has built-in immunity from `__repr__` looping perils covered ahead.

Listing instance attributes with __dict__

Let's get started with the simple case—listing attributes attached to an instance. Example 31-10, coded in the file *listinstance.py*, defines a mix-in called `ListInstance` that overloads the `__str__` method for all classes that include it in their header lines. Because this is coded as a class, `ListInstance` is a generic tool whose formatting logic can be used for instances of any subclass client.

Example 31-10. listinstance.py

```
class ListInstance:
    """
```

```
        Mix-in class that provides a formatted print() or s
        inheritance of __str__ coded here.  Displays instar
        instance of lowest class; __X naming avoids clashir
        Works for classes with slots: a __dict__ is ensurec
        """
        def __attrnames(self):
            result = '\n'
            for attr in sorted(self.__dict__):
                result += f'\t{attr}={self.__dict__[attr]!r
            return result

        def __str__(self):
            return (f'<Instance of {self.__class__.__name__
                    f'address {id(self):#x}:'
                    f'{self.__attrnames()}>')

    if __name__ == '__main__':
        import testmixin
        testmixin.tester(ListInstance)      # Test class ir
```

The `__attrnames` method here exhibits a classic comprehension pattern, and you might save some program real estate by implementing it more concisely with a *generator expression* triggered by a `''.join()` call; we'll leave this as a suggested exercise. As coded, `ListInstance` uses some previously explored techniques to extract the instance's class name and attributes:

- It uses a `self.__class__.__name__` expression to fetch the name of an instance's class. Recall that each instance has a built-in `__class__` attribute that references the class from which it was created, and each class has a `__name__` attribute that references the class's name given in its header line.
- It does most of its work by simply scanning the instance's attribute dictionary (remember, it's available in `__dict__`) to build up a string showing the names and values of all instance attributes. The dictionary's keys are sorted by name to be human-friendly, instead of relying on the dictionary's insertion order.

In these respects, `ListInstance` is similar to Chapter 28's attribute display; in fact, it's largely just a variation on a theme. Our class here, though, uses two additional techniques:

- It displays the instance's memory address by calling the `id` built-in function, which returns any object's address. By definition, this is a unique object identifier, which will be useful in later mutations of this code.
- It uses the *pseudoprivate* naming pattern for its worker method: `__attrnames`. As we saw earlier in this chapter, Python automatically localizes any such name to its enclosing class by expanding the attribute name to include the class name; in this case, it becomes `_ListInstance__attrnames`. This holds true for both class attributes like methods and instance attributes attached to `self`. As first noted in Chapter 28, this helps in a general tool like this, as it ensures that its names won't clash with any names used in its client subclasses.

Because `ListInstance` defines a `__str__` operator-overloading method, instances derived from this class display their attributes automatically when printed, giving a bit more information than a simple address. Here is this tool class in action in single-inheritance mode, mixed in to the previous section's class:

```
>>> from listinstance import ListInstance
>>> class Hack(ListInstance):                    # Inhe
        def __init__(self, what):
            self.data1 = what

>>> X = Hack('code')
>>> print(X)                                     # prir
<Instance of Hack, address 0x10c890b90:
        data1='code'
>
```

You can also get the listing display as a string with `str` without printing it (use `print` later to apply escapes), and interactive echoes and nesting still use the default format (we've left `__repr__` as an option for clients):

```
>>> str(X)
"<Instance of Hack, address 0x10c890b90:\n\tdata1='code
>>> X
<__main__.Hack object at 0x10c890b90>
```

The `ListInstance` class is useful for any classes you write—even classes that already have one or more superclasses. This is where *multiple inheritance* comes in handy: by adding `ListInstance` to the list of superclasses in a class header (i.e., mixing it in), you get its `__str__` "for free" while still inheriting from the existing superclass(es). The file *testmixin.py* in Example 31-11 codes test classes that prove the point.

Example 31-11. testmixin.py

```
"""
Generic lister-mixin tester: similar to transitive relo
Chapter 25, but passes a class object to tester (not fu
and testByNames adds loading of both module and class b
strings here, in keeping with Chapter 31's factories pa
"""
import importlib

def tester(listerclass, sept=False):
    "Pass any lister class to listerclass"

    class Super:
        def __init__(self):                 # Superclas
            self.data1 = 'code'             # Create ir
        def method1(self):
            pass

    class Sub(Super, listerclass):          # Mix in me
        def __init__(self):                 # Listers h
            Super.__init__(self)            # Or super(
            self.data2 = 'Python'           # More inst
            self.data3 = 3.12
        def method2(self):                  # Define ar
            pass

    instance = Sub()                        # Build ins
    print(instance)                         # Run mixed
    if sept: print(f'\n{'-' * 80}\n')       # Python 3.

def testByNames(modname, classname, sept=False):
    modobject   = importlib.import_module(modname)    #
    listerclass = getattr(modobject, classname)       #
    tester(listerclass, sept)

if __name__ == '__main__':
```

```
    testByNames('listinstance',  'ListInstance',  True)
    testByNames('listinherited', 'ListInherited', True)
    testByNames('listtree',      'ListTree',      False
```

This file is instrumented to allow us to test a variety of listers. To this end, it is passed a lister class, `listerclass`, to be mixed in with test classes nested in a function. Again, classes are passable first-class objects, and we need to make the lister class a parameter to allow it to vary per test. This file also has tools to fetch classes by name strings, which we'll set aside for the moment.

More important here is the self-test code in Example 31-10: it imports the `tester` function in Example 31-11 and passes in `ListInstance` to be tested here. Hence, `Sub` here inherits names from both `Super` and `ListInstance`; it's a *composite* of its own names and names in both its superclasses. When you make a `Sub` instance and print it, you automatically get the custom representation mixed in from `ListInstance`:

```
$ python3 listinstance.py
<Instance of Sub, address 0x10caae300:
        data1='code'
        data2='Python'
        data3=3.12
>
```

The `ListInstance` class responsible for this display works in any class it's mixed into because `self` refers to an instance of the subclass that pulls this class in, whatever that may be. Again, in a sense, mix-in classes are the class equivalent of modules—packages of methods useful in a variety of clients.

Besides the utility they provide, mix-ins optimize code maintenance, as all classes do. For example, if you later decide to extend `ListInstance`'s `__str__` to also print all the class attributes that an instance inherits, you're covered; because it's an inherited method, changing `__str__` automatically updates the display of each subclass that imports the class and mixes it in. And since it's now officially "later," let's move on to the next section to see what such an extension might look like.

Listing inherited attributes with dir

As it is, our `ListInstance` mix-in displays instance attributes only—names attached to the instance object itself. It's nearly trivial, though, to extend the class to display *all* the attributes accessible from an instance—both its own and those it inherits from its classes. The trick is to use the `dir` built-in function instead of scanning the instance's `__dict__` dictionary; the latter holds instance attributes only, but the former also collects all inherited attributes.

The mutation in Example 31-12, *listinherited.py*, codes this scheme. This is located in its own module to facilitate testing, but if existing clients were to use this version instead, they would pick up the new display automatically (and recall from Chapter 25 that a `from` import's `as` clause can rename a new version to a prior name being used).

Example 31-12. listinherited.py

```
class ListInherited:
    """
    Use dir() to collect both instance attrs and names
    its classes.  This includes default names inheritec
    'object' superclass above topmost classes.  getattr
    inherited names not in self.__dict__.

    Caution: use __str__, not __repr__, or else this lc
    bound methods that may be returned for some attribu
    This will normally fail for class "slots" attribute
    """

    def __attrnames(self, unders=False):
        result = '\n'
        for attr in dir(self):
            if attr[:2] == '__' and attr[-2:] == '__':
                result += f'\t{attr}\n' if unders else
            else:
                result += f'\t{attr}={getattr(self, att
        return result

    def __str__(self):
        return (f'<Instance of {self.__class__.__name__
                f'address {id(self):#x}:'
                f'{self.__attrnames()}>')
```

```
if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListInherited)      # Test class i
```

Notice that this code, by default, skips all `__X__` names for brevity. There are 27 such names in the test case, and most of these are internal names that we don't generally care about in a generic listing like this. Some are user-defined operator-overloading methods like our `__str__`, though most reflect defaults in the implicit `object` root class, and there's no easy way to determine an attribute's class of origin here (but stay tuned).

Because `dir` results may include names from anywhere in a class tree, this version also must use the `getattr` built-in function to fetch attributes by name string instead of indexing the instance's `__dict__` attribute dictionary. `getattr` runs inheritance search, and some of the names we're listing here are not stored on the instance itself.

To test the new version, run its file directly—it passes the `ListInherited` class it defines to the *testmixin.py* file's test function in Example 31-11 to be mixed in with a subclass in the function. Here's the output of this test and lister class; notice the extra names inherited from classes and the name mangling at work in the lister's method name:

```
$ python3 listinherited.py
<Instance of Sub, address 0x101d76600:
        _ListInherited__attrnames=<bound method ListInh
        data1='code'
        data2='Python'
        data3=3.12
        method1=<bound method tester.<locals>.Super.met
        method2=<bound method tester.<locals>.Sub.methc
>
```

The display of bound methods in this was truncated to fit this page; here's what the first two look like with an added line break (as usual, run on your own for the full and up-to-date picture):

```
        _ListInherited__attrnames=<bound method ListInh
            <testmixin.tester.<locals>.Sub object at 0x
```

```
method1=<bound method tester.<locals>.Super.met
    <testmixin.tester.<locals>.Sub object at 0x
```

Display formatting is an open-ended task (e.g., Python's standard `pprint` "pretty printer" module may offer options here too), so we'll leave further polishing as a suggested exercise. The tree lister of the next section may be more useful in any event, so let's move on.

NOTE

Looping in `__repr__` : Now that we're displaying inherited methods too, we must use `__str__` instead of `__repr__` to overload printing. With `__repr__` , this code (and code like it) will fall into *recursive loops*—its `getattr` returns a bound method; whose display includes the instance; which triggers `__repr__` again to display the instance; and so on, quickly triggering a stack-overflow exception. Subtle, but true! Change `__str__` to `__repr__` to see this live. One way to avoid such `__repr__` loops is to skip `getattr` results for which `isinstance` comparisons to the standard library's `types.MethodType` are true. Using `__str__` instead is simpler.

Listing attributes per object in class trees

Let's code one last extension. As it is, our latest lister includes inherited names but doesn't give any sort of designation of the classes from which the names are acquired. As we saw in the *classtree.py* example near the end of Chapter 29, though, it's straightforward to climb class inheritance trees in code.

The mix-in class in Example 31-13, coded in file *listtree.py*, makes use of this same technique to display attributes grouped by the classes in which they live —it sketches the full *physical class tree*, displaying attributes attached to each object along the way. The reader must still infer attribute inheritance (and we'll address this in the next section's final demo), but this version gives substantially more detail than a simple flat list of attributes, inherited or not.

Example 31-13. listtree.py

```
class ListTree:
    """
    Mix-in that returns a __str__ trace of the entire c
```

```
its objects' attrs at and above the self instance.
run by print() automatically; use str() to fetch as

-Uses __X pseudoprivate attr names to avoid conflic
-Recurses to superclasses explicitly in DLFR (thoug
-Uses __dict__ instead of dir() because attrs are l
-Supports classes with slots: lack of slots here er
"""

def __attrnames(self, obj, indent, unders=True):
    spaces = ' ' * (indent + 1)
    result = ''
    for attr in sorted(obj.__dict__):
        if attr.startswith('__') and attr.endswith(
            if unders: result += f'{spaces}{attr}\r
        else:
            result += f'{spaces}{attr}={getattr(obj
    return result

def __listclass(self, aClass, indent):
    dots = '.' * indent
    preamble = (f'\n{dots}'
                f'<Class {aClass.__name__}'
                f', address {id(aClass):#x}')

    if aClass in self.__visited:
        return preamble + ': (see above)>\n'
    elif aClass is object:
        self.__visited[aClass] = True
        return preamble + ': (see dir(object))>\n'
    else:
        self.__visited[aClass] = True
        here  = self.__attrnames(aClass, indent)
        above = ''
        for Super in aClass.__bases__:
            above += self.__listclass(Super, indent
        return preamble + f':\n{here}{above}{dots}>

def __str__(self):
    self.__visited = {}
    here  = self.__attrnames(self, 0)
    above = self.__listclass(self.__class__, 4)
    return (f'<Instance of {self.__class__.__name__
            f', address {id(self):#x}'
            f':\n{here}{above}>')
```

```
if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListTree)      # Test class in thi
```

This class achieves its goal by traversing the inheritance tree—from an instance's `__class__` to its class, and then from the class's `__bases__` to all superclasses recursively, scanning each object's attribute `__dict__` along the way to enumerate attributes. Ultimately, it concatenates each tree portion's string as the recursion unwinds.

It can take a few moments to understand recursive programs like this, but given the arbitrary shape and depth of class trees, we really have no choice here (apart from explicit stack equivalents of the sorts we met in Chapters 19 and 25, which tend to be no simpler, and which we'll omit here for space and time). This class is coded to keep its business as explicit as possible, though, to maximize clarity.

To test, run this class's module file as before; it passes the `ListTree` class to *testmixin.py* of Example 31-11 again, to be mixed in with a subclass in the test function. The file's tree-sketcher output is as follows:

```
$ python3 listtree.py
<Instance of Sub, address 0x10a45f980:
 _ListTree__visited={}
 data1='code'
 data2='Python'
 data3=3.12

....<Class Sub, address 0x7fb26a448530:
     __doc__
     __init__
     __module__
     method2=<function tester.<locals>.Sub.method2 at 0

........<Class Super, address 0x7fb26a445c00:
         __dict__
         __doc__
         __init__
         __module__
         __weakref__
         method1=<function tester.<locals>.Super.methoc
```

```
............<Class object, address 0x10a01b100: (see di
........>

........<Class ListTree, address 0x7fb26a443d20:
          _ListTree__attrnames=<function ListTree.__attr
          _ListTree__listclass=<function ListTree.__list
          __dict__
          __doc__
          __module__
          __str__
          __weakref__

............<Class object, address 0x10a01b100: (see at
........>
....>
>
```

Some points to notice about this example:

- The `__visited` table's name is mangled in the instance's attribute dictionary for *pseudoprivacy*; unless we're very unlucky, this won't clash with other data there. Some of the lister class's methods are mangled as well.
- To minimize displays, `__X__` attributes are listed by *name* only, skipping their values. The built-in `object` class implied above all topmost classes is also singled out to simply refer readers to its `dir` result; `object` comes with 24 attributes today, which wouldn't be useful to repeat in every display.
- The attributes that were bound methods in the prior version are now plain *functions*. This reflects the fact that this version fetches methods from *classes* instead of the instance—the `getattr` here is run on the current tree object whose `__dict__` is being scanned (`getattr` and `__dict__` indexing are equivalent in this context). Again, class methods are just functions, which are bound only when fetched from an instance.
- To avoid listing a class object more than once, a table records classes *visited* so far. A dictionary works in this role because class objects are hashable and thus may be dictionary keys; a set would work similarly.

On the last point, *cycles* are not generally possible in class inheritance trees—a class must already have been defined to be named as a superclass, and Python raises an exception as it should if you attempt to create a cycle later by

`__bases__` changes. The visited mechanism here is still needed, though, to avoid relisting a class twice:

```
>>> class C: pass
>>> class B(C): pass
>>> C.__bases__ = (B,)        # Dark magic
TypeError: a __bases__ item causes an inheritance cycle
```

For more fun, try mixing this class into something more substantial, like the `Button` class of Python's `tkinter` GUI-toolkit module. In general, you'll want to name `ListTree` first (*leftmost*) in a `class` header, so its `__str__` is picked up; `Button` has one, too, and the leftmost superclass is searched first in multiple inheritance's default:

```
>>> from tkinter import Button
>>> from listtree import ListTree

>>> class ButtonPlus(ListTree, Button): pass        # Li
>>> print(ButtonPlus())
…our class's display…

>>> class ButtonPlus(Button, ListTree): pass        # Mi
>>> print(ButtonPlus())
.!buttonplus2
```

Order matters in multiple inheritance, though the manual overrides we explored earlier can force the issue:

```
>>> class ButtonPlus(Button, ListTree): __str__ = ListT
…our class's display…
```

You might also try running *testmixin.py* of Example 31-11 directly; its self-test code that we shelved earlier runs each of our three lister classes in turn, using their module and class name strings—a trivial class factory in action:

```
$ python3 testmixin.py
…all three listers' results…
```

While our tree lister works as planned, it doesn't really follow Python's MRO ordering through class trees with diamonds. In fact, it really just sketches the DFLR order and leaves it to readers to determine from which class a given attribute is inherited. To do better, let's move on to a final example to close out this chapter.

Example: Mapping Attributes to Inheritance Sources

This section wraps up with an example that demos an application for the MRO in programming tools. As coded, the preceding section's tree lister gave the *physical* locations of attributes in a class tree. However, by mapping the list of inherited attributes in a `dir` result to the linear MRO sequence, such tools can more directly associate attributes with the classes from which they are actually *inherited*—also a useful relationship for programmers.

We won't recode our tree lister in full here, but as a first major step, Example 31-14, file *mapattrs.py*, implements tools that can be used to associate attributes with their inheritance source. As an added bonus, its `mapattrs` function demonstrates how inheritance actually searches for attributes in class tree objects—though the MRO is largely automated for us with the built-in `__mro__` class attribute we met earlier.

Example 31-14. mapattrs.py

```
"""
Main tool: mapattrs() maps all attributes on or inherit
instance to the instance or class from which they are i
Also here: assorted dictionary tools using comprehensic

Assumes dir() gives all attributes of an instance.  To
inheritance, this uses the class's __mro__ tuple, which
MRO search order for classes in Python 3.X.  A recursiv
traversal for the DFLR order of classes is included but
"""

import pprint
def trace(label, X, end='\n'):
    print(f'{label}\n{pprint.pformat(X)}{end}')   # Pri

def filterdictvals(D, V):
    """
    dict D with entries for value V removed.
```

```
    filterdictvals(dict(a=1, b=2, c=1), 1) => {'b': 2}
    """
    return {K: V2 for (K, V2) in D.items() if V2 != V}

def invertdict(D):
    """
    dict D with values changed to keys (grouped by valu
    Values must all be hashable to work as dict/set key
    invertdict(dict(a=1, b=2, c=1)) => {1: ['a', 'c'],
    """
    def keysof(V):
        return sorted(K for K in D.keys() if D[K] == V)
    return {V: keysof(V) for V in set(D.values())}

def dflr(cls):
    """
    Depth-first left-to-right order of class tree at cl
    Cycles not possible: Python disallows on __bases__
    """
    here = [cls]
    for sup in cls.__bases__:
        here += dflr(sup)
    return here

def inheritance(instance):
    """
    Inheritance order sequence: MRO or DFLR.
    DFLR alone is no longer used in Python 3.X.
    """
    if hasattr(instance.__class__, '__mro__'):
        return (instance,) + instance.__class__.__mro__
    else:
        return [instance] + dflr(instance.__class__)

def mapattrs(instance, withobject=False, bysource=False
    """
    dict with keys giving all inherited attributes of i
    with values giving the object that each is inherite
    withobject: False=remove object built-in class attr
    bysource:   True=group result by objects instead of
    Supports classes with slots that preclude __dict__
    """
    attr2obj = {}
    inherits = inheritance(instance)
    for attr in dir(instance):
        for obj in inherits:
```

```
                if hasattr(obj, '__dict__') and attr in ob
                    attr2obj[attr] = obj
                    break

        if not withobject:
            attr2obj = filterdictvals(attr2obj, object)
        return attr2obj if not bysource else invertdict(att

    if __name__ == '__main__':

        class D:         attr2 = 'D'
        class C(D):      attr2 = 'C'
        class B(D):      attr1 = 'B'
        class A(B, C):   pass
        I = A()
        I.attr0 = 'I'

        print(f'Py=>{I.attr0=}, {I.attr1=}, {I.attr2=}\n')
        trace('INHERITANCE', inheritance(I))
        trace('ATTRIBUTES',  mapattrs(I))
        trace('SOURCES',     mapattrs(I, bysource=True))
```

This module's main `mapattrs` function uses `dir` to collect all the attributes that an instance inherits. For each, it maps the attribute to its source by scanning the MRO order available in the `__mro__` of the instance's class, searching each object's namespace `__dict__` along the way. The net effect replicates Python's true inheritance search for each attribute accessible from the instance passed in.

This file's self-test code applies its tools to a diamond multiple-inheritance tree similar to those we studied earlier. It uses Python's `pprint` standard-library module to display lists and dictionaries nicely— `pprint.pprint` is its basic call, and its `pformat` returns a print string. Notably, `attr2`, whose value is given on the first line and whose name appears in later function results, is inherited from class `C` per the MRO order we've studied:

```
$ python3 mapattrs.py
Py=>I.attr0='I', I.attr1='B', I.attr2='C'

INHERITANCE
(<__main__.A object at 0x10cc33e00>,
 <class '__main__.A'>,
 <class '__main__.B'>,
```

```
 <class '__main__.C'>,
 <class '__main__.D'>,
 <class 'object'>)

ATTRIBUTES
{'__dict__': <class '__main__.D'>,
 '__doc__': <class '__main__.A'>,
 '__module__': <class '__main__.A'>,
 '__weakref__': <class '__main__.D'>,
 'attr0': <__main__.A object at 0x10cc33e00>,
 'attr1': <class '__main__.B'>,
 'attr2': <class '__main__.C'>}

SOURCES
{<__main__.A object at 0x10cc33e00>: ['attr0'],
 <class '__main__.D'>: ['__dict__', '__weakref__'],
 <class '__main__.C'>: ['attr2'],
 <class '__main__.B'>: ['attr1'],
 <class '__main__.A'>: ['__doc__', '__module__']}
```

Although this module was not designed to be a mix-in class itself, listers may index its `mapattrs` function's dictionary results to obtain an attribute's source or a source's attributes. Moreover, it's easy to adapt this module's results to be a mix-in by wrapping them in a `__str__`. Here it is listing attributes' sources in the test classes of the prior section's Example 31-11:

```
$ python3
>>> import pprint
>>> from mapattrs import mapattrs
>>> class ListAttr2Source:
        def __str__(self):
            return pprint.pformat(mapattrs(self))

>>> from testmixin import tester
>>> tester(ListAttr2Source)
{'__dict__': <class 'testmixin.tester.<locals>.Super'>,
 '__doc__': <class 'testmixin.tester.<locals>.Sub'>,
 '__init__': <class 'testmixin.tester.<locals>.Sub'>,
 '__module__': <class 'testmixin.tester.<locals>.Sub'>,
 '__str__': <class '__main__.ListAttr2Source'>,
 '__weakref__': <class 'testmixin.tester.<locals>.Super
 'data1': <testmixin.tester.<locals>.Sub object at 0x10
 'data2': <testmixin.tester.<locals>.Sub object at 0x10
 'data3': <testmixin.tester.<locals>.Sub object at 0x10
```

```
 'method1': <class 'testmixin.tester.<locals>.Super'>,
 'method2': <class 'testmixin.tester.<locals>.Sub'>}
```

Listing sources' attributes is just as easy (see also Python's docs for `pprint` options like `compact` and `width`):

```
>>> class ListSource2Attr:
        def __str__(self):
            return pprint.pformat(mapattrs(self, bysour

>>> tester(ListSource2Attr)
{<testmixin.tester.<locals>.Sub object at 0x102ad12e0>:


 <class 'testmixin.tester.<locals>.Super'>: ['__dict__'
                                             '__weakref
                                             'method1']
 <class 'testmixin.tester.<locals>.Sub'>: ['__doc__',
                                           '__init__',
                                           '__module__'
                                           'method2'],
 <class '__main__.ListSource2Attr'>: ['__str__']}
```

Study this example's code for more insight. As callouts, notice how it uses `hasattr` to check whether an object has a `__dict__` attribute dictionary before trying to index it. Though rare, some instances may not have a `__dict__` if they use the class extension known as *slots* noted earlier. The prior section's slot story is varied: as mix-ins, `ListTree` and `ListInstance` work as is for classes with slots because their *lack* of slots ensures an instance `__dict__`, but `ListInherited` can fail for slots not yet assigned—findings to be clarified in the next chapter.

Additionally, both slots and other "virtual" instance attributes like *properties* and *descriptors* live at the class instead of the instance and hence may require generic handling— `dir` enumeration, and either `getattr` fetches, tree climbs, or MRO scans. This example and the prior section's listers accommodate this, but unevenly: some such names will be associated with the classes in which their implementations live, not the instance through which they are accessed.

Moreover, no lister can show attribute names dynamically computed in full by methods like `__getattr__` because these names have no physical basis. Classes implementing such dynamic names can also define a `__dir__` method to provide an attribute result list for `dir` calls, but general tools like our listers and mapper cannot depend on this optional and relatively uncommon interface being present.

Finally, all the attribute listers and mappers in this chapter work in full for normal *instances* but don't support *classes*. For the latter, prints run a default display instead of any of the three listers, and `mapattrs` strangely attributes most names to a mystery class called "type." The lister skips stem from the fact that built-ins like `print` skip the "instance," as we've noted before. The `mapattrs` oddity reflects the fact that classes acquire names from both their own superclass tree (and MRO), and a secondary tree (and MRO) formed by "metaclasses" that we have yet to meet.

But to understand both the inheritance bifurcation of metaclasses, as well as ethereal attributes like slots, properties, and descriptors, we need to move on to the next chapter.

Other Design-Related Topics

In this chapter, we've studied an assortment of design patterns used to combine classes in Python programs, along with the mechanism behind some of them. We've really only scratched the surface here in the design patterns domain, though. Elsewhere in this book, you'll find coverage of other design-related topics, such as:

- *Abstract superclasses* (Chapter 29)
- *Decorators* (Chapters 32 and 39)
- *Type subclasses* (Chapter 32)
- *Static and class methods* (Chapter 32)
- *Managed attributes* (Chapters 32 and 38)
- *Metaclasses* (Chapters 32 and 40)

For even more details on design patterns, this book must delegate to other resources on OOP at large. Although patterns are important in OOP work and are often more natural in Python than other languages, they are not specific to Python itself and a subject that's often best acquired by experience.

Chapter Summary

In this chapter, we sampled common ways to use and combine classes to optimize their reusability and factoring benefits—what are usually considered design issues, which are often independent of any particular programming language (though Python can make them easier to implement). We studied *inheritance* (acquiring behavior from other classes), *composition* (controlling embedded objects), and *delegation* (wrapping objects in proxy classes), as well as the related topics of pseudoprivate attributes, bound methods, factories, multiple inheritance, and the MRO.

The next chapter concludes our look at classes and OOP by surveying class-related topics that are more esoteric than most of what we've already seen. Some of its material may be of more interest to tool writers than application programmers, but it still merits a review by most people who will do OOP in Python—if not for your code, then for others' code you may need to understand and reuse. First, though, here's another quick chapter quiz to review.

Test Your Knowledge: Quiz

1. What is multiple inheritance?
2. What is composition?
3. What is delegation?
4. What are bound methods?
5. What are pseudoprivate attributes used for?
6. How does the MRO inheritance search order differ from DFLR?

Test Your Knowledge: Answers

1. *Multiple inheritance* occurs when a class inherits from more than one superclass; it's useful for mixing together multiple packages of class-based code. The left-to-right order in `class` statement headers determines the general order of attribute searches, and the MRO specializes search for diamonds with common superclasses.
2. *Composition* is a technique whereby a controller class embeds and directs a number of objects and provides an interface all its own; it's a way to

build up larger structures with classes.

3. *Delegation* involves wrapping an object in a proxy class, which adds extra behavior and passes other operations to the wrapped object. The proxy generally retains the interface of the wrapped object.
4. *Bound methods* combine an instance and a method function; you can call them without passing in an instance object explicitly because the original instance is still available in the instance+function pair.
5. *Pseudoprivate attributes*, whose names begin but do not end with two leading underscores (e.g., `__X`), are used to localize names to the enclosing class. This includes both class attributes, like methods defined inside the `class` statement, and `self` instance attributes assigned inside the class's methods. Such names are expanded to include the class name, which makes them generally unique among all classes in an inheritance tree.
6. The MRO selects same-named attributes in a *lower* subclass over those in a higher common superclass in multiple-inheritance "diamond" trees—effectively searching across before up in this specific case. The DFLR is otherwise the same; in fact, the MRO is defined by *starting* with the DFLR order and then *removing* all but the last (rightmost) appearances of classes that are visited more than once. This differs from DFLR only when there are duplicates, which arise only in diamonds that have common superclasses. That said, the built-in `object` makes every multiple-inheritance tree a diamond, so this is a common, if implicit, occurrence.