Chapter 40. Metaclasses and Inheritance

In Chapter 39, we explored decorators and studied examples of their use. In this final technical chapter of the book, we're going to continue our tool-builders focus with an in-depth review of another advanced topic: *metaclasses*, a protocol for managing class objects instead of their instances, introduced briefly in Chapter 32.

On a base level, metaclasses extend the code-insertion model of decorators. As we learned in the prior chapter, decorators allow us to augment functions and classes by intercepting their creation. Metaclasses similarly allow us to intercept and augment *class creation*—they provide a hook for inserting extra logic to be run at the conclusion of a `class` statement, albeit in different ways than decorators.

Metaclasses can also provide behavior for classes with *methods* located in a separate inheritance tree skipped for normal, nonclass instances. While this allows metaclasses to process their instance classes after creation, it also compounds class semantics and convolutes *inheritance*—whose full definition can finally be fleshed out here.

Like all the subjects covered in this part of the book, this is an *advanced topic* that can be studied on an as-needed basis. Metaclasses are not generally in scope for most application programmers but may be of interest to others seeking to write flexible tools. Whatever category you fall into, though, metaclasses can teach you more about Python's classes and are a prerequisite to both code that employs them and the complete inheritance story in Python.

As the last technical chapter of this book, this also begins to wrap up some threads concerning Python itself that we have met often along the way and will finalize in the conclusion that follows. Where you go after this book is up to you, but in an open source project, it's important to keep the big picture in mind while hacking the small details.

To Metaclass or Not to Metaclass

Despite the advanced status awarded to metaclasses in the preceding opener, they have a variety of potential roles. For example, they can be used to enhance classes with features like tracing, object persistence, exception logging, and more. They can also be used to construct portions of a class at runtime based on configuration files, apply function decorators to every method of a class generically, verify conformance to expected interfaces, and so on.

In their more grandiose incarnations, metaclasses can even be used to implement alternative coding patterns such as aspect-oriented programming, object/relational mappers (ORMs) for databases, and more. Although there are often alternative ways to achieve such results—as you'll see, the roles of class decorators and metaclasses often intersect—metaclasses provide a formal model tailored to those tasks. We don't have space to explore all such applications first-hand in this chapter, of course, but you can find additional use cases on the web after studying the basics here.

Probably the reason for studying metaclasses most relevant to this book is that this topic can help demystify Python's class mechanics in general. For instance, you'll find that they are an intrinsic part of the language's inheritance model formalized in full here. Although you may or may not code or reuse them in your work, a cursory understanding of metaclasses can impart a deeper understanding of Python at large.

The Downside of "Helper" Functions

Before we get to metaclass code, let's get a better handle on its rationale. Like the decorators of the prior chapter, metaclasses are optional in principle. We can usually achieve the same effect by passing class objects through functions —known interchangeably as *helper* or *manager* functions—much as we can achieve the goals of decorators by passing functions and classes through manager code. Just like decorators, though, metaclasses:

- Provide a more uniform and explicit structure
- Help ensure that application programmers won't forget to augment their classes according to an API's requirements

- Avoid code redundancy and its associated maintenance costs by factoring class customization logic into a single location

To illustrate, suppose we want to automatically insert a method into a set of classes. Of course, we could do this with simple *inheritance* if the subject method is known when we code the classes. In that case, we can simply code the method in a superclass and have all the classes in question inherit from it:

```
class Extras:
    def extra(self, args):              # Normal inheri
        …

class Client1(Extras): …                # Clients inher
class Client2(Extras): …

X = Client1()                           # Make an insta
X.extra()                               # Run the extra
```

Sometimes, though, it's impossible to predict such augmentation when classes are coded. Consider the case where classes are augmented in response to choices made in a user interface at runtime or loaded from an editable configuration file. Although we could code every class in our imaginary set to *manually* check these, too, it's a lot to ask of clients (the `required` function here is abstract—it's something to be filled in):

```
def extra(self, arg): …

class Client1: …                        # Client augmer
if required():
    Client1.extra = extra

class Client2: …
if required():
    Client2.extra = extra               # Add the extra

X = Client1()
X.extra()
```

We can add methods to a class after the `class` statement like this because, as we've learned, a class-level method is just a plain function that is

associated with a class and has a first argument to receive a `self` instance when called through one. Although this works, it might become untenable for larger method sets and puts all the burden of augmentation on each client class (and assumes they'll remember to do this at all).

It would be better from a maintenance perspective to isolate the decision logic in a single place. We might encapsulate some of this extra work by routing classes through a *helper function*—a function that would extend the class as required and handle all the work of runtime testing and configuration:

```
def extra(self, arg): …

def extras(Class):                          # Helper functi
    if required():
        Class.extra = extra

class Client1: …
extras(Client1)

class Client2: …
extras(Client2)

X = Client1()
X.extra()
```

This code runs the class through a helper function immediately after it is created. Although functions like this one can achieve our goal here, they still put a burden on class coders, who must understand the requirements and adhere to them in their code. It would be even better if there was a simple way to enforce the augmentation in the subject classes, so that they don't need to deal with the augmentation so explicitly and would be less likely to forget to use it altogether. In other words, we'd like to be able to insert some code to run *automatically* at the end of a `class` statement to augment the class.

This is exactly what *metaclasses* do—by declaring a metaclass, we tell Python to route the creation of the class object to another class we provide:

```
def extra(self, arg): …

class Extras(type):
    def __init__(Class, classname, superclasses, attrib
```

```
            if required():
                Class.extra = extra

    class Client1(metaclass=Extras): …      # Metaclass dec
    class Client2(metaclass=Extras): …      # Client class

    X = Client1()                           # X is instance
    X.extra()
```

Because Python invokes the metaclass automatically at the end of the `class` statement when the new class is *created*, it can augment, register, wrap, or otherwise manage the class as needed. Moreover, the only requirement for the client classes is that they *declare* their metaclass; every class that does so will automatically acquire whatever augmentation the metaclass provides, both now and in the future if the metaclass changes.

Metaclasses can also augment their class instances with inherited *methods*, which are akin to normal class methods but not inherited by the instances of their class instances—an inheritance extension and tongue twister whose true nature requires more info ahead (and quite possibly, sedation). Through both changes and methods, though, metaclasses can customize class behavior broadly.

Of course, this is the standard rationale, which you'll need to judge for yourself—in truth, clients might forget to list a metaclass just as easily as they could forget to call a helper function! Still, the explicit nature of metaclasses may make this less likely. Although it may be difficult to glean from this small and hypothetical example, metaclasses generally handle such tasks better than more manual approaches.

Metaclasses Versus Class Decorators: Round 1

Having said that, it's also important to note that the *class decorators* described in the preceding chapter sometimes overlap with metaclasses—in terms of both utility and benefit. Like metaclasses, class decorators can be used to manage both classes and their later instances, and their syntax makes their usage similarly explicit and arguably more obvious than helper-function calls.

For example, suppose we recoded the last section's helper function to *return* the augmented class instead of simply modifying it in place. This would allow

a greater degree of flexibility because the manager would be free to return any type of object that implements the class's expected interface:

```
def extra(self, arg): …

def extras(Class):
    if required():
        Class.extra = extra
    return Class                # Return the augmented c

class Client1: …
Client1 = extras(Client1)       # Rebind to augmented cl

class Client2: …
Client2 = extras(Client2)

X = Client1()
X.extra()
```

If you think this is starting to look reminiscent of class decorators, you're right. In the prior chapter, we emphasized class decorators' role in augmenting *instance* creation calls. Because they work by automatically rebinding a class name to the result of a function, though, there's no reason that we can't use them to augment the class by changing it before any instances are ever created. That is, class decorators can apply extra logic to *classes*, not just *instances*, at class creation time:

```
def extra(self, arg): …

def extras(Class):              # From helper to decorat
    if required():
        Class.extra = extra
    return Class

@extras
class Client1: …                # Client1 = extras(Clier

@extras
class Client2: …                # Rebinds class indepenc
```

```
X = Client1()                    # Makes instance of augm
X.extra()                        # X is instance of origi
```

Decorators essentially automate the prior example's manual name rebinding here. Just as for metaclasses, because this decorator returns the original class, instances are made from that class, not from a wrapper object. In fact, instance creation is not intercepted at all in this example.

In this specific case—adding methods to a class when it's created—the choice between metaclasses and decorators is arbitrary. Decorators can be used to manage both instances and classes and intersect most strongly with metaclasses in the second of these roles, but this discrimination is not absolute. In fact, the roles of each are suggested in part by their mechanics.

As we'll detail ahead, decorators technically correspond to metaclass *calls* used to make and initialize new classes. Metaclasses, though, have additional customization hooks beyond class creation. Their *methods* inherited by classes, for example, have no direct counterpart in class decorators sans extra code. This can make metaclasses more complex but also better suited for augmenting classes in some contexts.

Conversely, because metaclasses are designed to manage classes, applying them to managing *instances* alone is less optimal. Because they are also responsible for making the class itself, metaclasses incur this as an *extra* step in instance-management roles and are perhaps less appropriate than decorators.

We'll explore some of these differences in working code later in this chapter. To better understand how metaclasses do their work, though, we first need to get a clearer picture of their underlying model.

The Metaclass Model

To understand metaclasses, you first need to understand a bit more about both Python's object model and what happens at the end of a `class` statement. As you'll learn here, the two are intimately related.

Classes Are Instances of type

The first of these prerequisites was covered previously by "The Python Object Model", which in turn assumes knowledge of the *MRO* (method resolution order) covered in Chapter 31. You should review that content now if needed, especially if you've jumped into this chapter at random. We're not going to repeat its coverage in full here, but some of its conclusions are crucial to understanding metaclasses. Namely:

- Instances are created from classes.
- Classes are instances of a metaclass.
- The `type` built-in is the topmost metaclass.
- Metaclasses customize `type` with normal `class` statements.

In short, classes are types, types are classes, and metaclasses customize types. Per Chapter 32, the relationship between metaclasses and classes is subtly different from that between classes and their *nonclass* instances. The latter do not generate more instances, and while attribute inheritance uses the same core mechanisms and MRO everywhere, classes search a metaclass tree that nonclass instances do not.

We'll study the nuts and bolts of inheritance ahead, but it's easy to see this model's fundamentals in code. Built-in objects like lists are actually nonclass instances made from a class, which itself is made from the built-in `type`:

```
>>> type([]), type(type([]))          # List instance is
(<class 'list'>, <class 'type'>)      # List class is cr

>>> type(list), type(type)            # Same, but with t
(<class 'type'>, <class 'type'>)      # Type of type is
```

Apart from the literal syntax of built-ins, this works the same way for user-defined classes, which are really just user-defined types—their nonclass instances are made from the class, which itself is made from the built-in `type`:

```
>>> class Hack: pass                  # User-defined cla
>>> I = Hack()                        # Made from class,
```

```
>>> type(I), type(type(I))
(<class '__main__.Hack'>, <class 'type'>)
```

Although their behavior varies in ways we explored in Chapter 32, both classes and nonclass instances are "instances" in some sense. In fact, the `__class__` attribute in both tells us what they were made from:

```
>>> [].__class__, list.__class__      # Instances and cl
(<class 'list'>, <class 'type'>)      # type(X) is normc

>>> I.__class__, Hack.__class__
(<class '__main__.Hack'>, <class 'type'>)
```

Because classes are created from the root `type` class by default, most programmers don't need to think about this model. However, it's key to understanding the way that metaclasses work—as the next section explains.

Metaclasses Are Subclasses of type

Why would we care that classes are instances of a `type` class? It turns out that this is the hook that allows us to code metaclasses. Specifically, we can create classes from *subclasses* of `type` that customize it with normal object-oriented techniques and class syntax. And these `type` subclasses are known as *metaclasses*.

In other words, to control the way classes are created and augment their behavior, all we need to do is specify that a user-defined class be created from a user-defined metaclass instead of the normal and default `type` class.

Before we see how, it's important to bear in mind that this *type instance* relationship is not quite the same as normal *inheritance*. User-defined classes may also have *superclasses* from which they and their instances inherit attributes as usual. As we've seen, inheritance superclasses are listed in parentheses in the `class` statement, show up in a class's `__bases__` tuple, and are searched for attributes fetched from nonclass instances.

However, the type from which a class is created and of which it is an instance is a different relationship. Inheritance searches instance and class namespace dictionaries, but classes may also acquire behavior from their type that is not

exposed to the normal inheritance search. In fact, metaclasses define a separate, *secondary* inheritance tree available only to classes and used as a fallback when the normal superclass search fails.

To lay the groundwork for understanding this distinction, the next section describes the procedure and syntax Python uses to implement this *instance-of* relationship.

Class Statements Call a type

Subclassing the `type` class to customize it is really only half of the metaclass backstory. We still need to somehow route a class's creation to the metaclass instead of the default `type` . To comprehend the way that this is arranged, we also need to know how `class` statements do their business.

We've already learned that when Python reaches a `class` statement, it runs its nested block of code to create the class's attributes—all the names assigned at the top level of the nested code block generate attributes in the resulting class object. These names are usually method functions created by nested `def` s, but they can also be arbitrary attributes assigned to create class data shared by all instances.

Technically speaking, Python follows a standard protocol to make this happen: at the *end* of a `class` statement, and after running all its nested code in a namespace dictionary corresponding to the class's local scope, Python calls the `type` object to create the new class object like this:

```
class = type(classname, superclasses, attributedict)
```

The `type` object in turn defines a `__call__` operator-overloading method that runs two other methods when the `type` object is called:

```
type.__new__(typeclass, classname, superclasses, attrib
type.__init__(class, classname, superclasses, attribute
```

The `__new__` method creates and returns the new `class` object, after which the `__init__` method initializes the newly created object. As you'll

see in a moment, these are the hooks that metaclass subclasses of `type` generally use to perform class customizations at creation time.

For example, given a class definition like the following for `Hack`:

```
class Super: …                          # Inherited names her

class Hack(Super):                      # Inherits from Super
    data = 1                            # Class data attribut
    def meth(self, arg):                # Class method attrib
        return self.data + arg
```

Python will internally run the nested code block to create two attributes of the class (`data` and `meth`), and then call the `type` object to generate the `class` object at the end of the `class` statement's processing (extra names like `__module__` are added automatically from the code's context):

```
Hack = type('Hack', (Super,), {'data': 1, 'meth': meth,
```

In fact, you can call `type` this way yourself to create a class *dynamically*—albeit here with a fabricated method function and empty superclasses tuple (Python adds the `object` superclass automatically to topmost classes as we learned in Chapter 32, and the enclosing module's name is again implied):

```
>>> c = type('Hack', (), {'data': 1, 'meth': (lambda x,
>>> i = c()
>>> c, i
(<class '__main__.Hack'>, <__main__.Hack object at 0x10
>>> i.data, i.meth(2)
(1, 3)
```

The class produced by a direct `type` call is exactly like that you'd get from running a `class` statement (again, if you've forgotten what some of the following are about, flip, click, or tap back to Chapter 32 for a refresher):

```
>>> c.__bases__
(<class 'object'>,)
>>> i.__class__.__mro__
```

```
(<class '__main__.Hack'>, <class 'object'>)

>>> [a for a in dir(i) if not a.startswith('__')]
['data', 'meth']

>>> [(a, v) for (a, v) in c.__dict__.items() if not a.s
[('data', 1), ('meth', <function <lambda> at 0x1082179c
```

Because this `type` call is made automatically at the end of the `class` statement, though, it's an ideal hook for augmenting or otherwise processing a class. The trick lies in replacing the default `type` with a custom subclass that will intercept this call. The next section shows how.

Class Statements Can Choose a type

As we've just seen, classes are created by the `type` class by default. To tell Python to create a class with a custom *metaclass* instead, you simply need to declare a metaclass to intercept the normal instance creation call for a user-defined class. To do so, list the desired metaclass as a keyword argument in the `class` header:

```
class Hack(metaclass=Meta):                    # Use Met
```

If no such declaration is present, the metaclass to be called defaults to the `type` built-in, per the prior section. When used, though, this declaration overrides the `type` default and routes the class creation call at the close of the `class` statement to `Meta` instead:

```
class = Meta(classname, superclasses, attributedict)
```

Importantly again, the `metaclass` keyword specifies an *instance-of* relationship, which implies inheritance only through the secondary metaclass tree we'll formalize ahead. Normal *inheritance* superclasses can be listed in the header as well and take precedence by residing in the primary class tree. In the following, for example, the new class `Hack` inherits from superclass `Super` normally but is also an instance of—and is created by—metaclass `Meta`:

```
class Hack(Super, metaclass=Meta):            # Normal
```

In this form, superclasses must be listed before the metaclass; in effect, the ordering rules used for keyword arguments in function calls apply here too. This order also has implications for inheritance, which we'll formalize soon, but first, we need to learn how to code the metaclasses that tap into calls triggered by this special `class` syntax.

Metaclass Method Protocol

When a specific metaclass is declared per the prior sections' syntax, the call to create the `class` object run at the end of the `class` statement is modified to invoke the *metaclass* instead of the `type` default, as we just saw:

```
class = Meta(classname, superclasses, attributedict)
```

Assuming the metaclass is a subclass of `type`, though, the `type` class's inherited `__call__` method delegates creation and initialization of the new `class` object to the metaclass if the metaclass defines custom versions of the methods that handle these steps. In other words, the `Meta` call may wind up triggering these method calls in turn:

```
class = Meta.__new__(Meta, classname, superclasses, att
Meta.__init__(class, classname, superclasses, attribute
```

To demonstrate, here's the preceding class example again, augmented with a metaclass specification:

```
class Hack(Super, metaclass=Meta):      # Inherits from
    data = 1                            # Class data att
    def meth(self, arg):                # Class method c
        return self.data + arg
```

At the end of this `class` statement, Python internally runs the following to create the `class` object—again, a call you could make manually, too, but automatically run by Python's `class` machinery:

```
Hack = Meta('Hack', (Super,), {'data': 1, 'meth': meth,
```

If the metaclass defines its own versions of `__new__` or `__init__`, they will be invoked during this call by the inherited `type` class's `__call__` method. The net effect is to automatically run methods the metaclass provides as part of the class-construction process. The next section shows how we might go about coding this final piece of the metaclass puzzle.

Coding Metaclasses

So far, we've seen how Python routes class creation calls to a metaclass if one is specified and provided. How, though, do we actually *code* a metaclass that customizes `type`?

It turns out that you already know most of the story—metaclasses are coded with normal Python `class` statements and semantics. By definition, they are simply classes that inherit from `type` (normally, at least). Their only substantial distinctions are that Python calls them *automatically* at the end of a `class` statement and that they must generally adhere to the *interface* expected by the `type` superclass if they subclass it.

A Basic Metaclass

Perhaps the simplest metaclass you can code is simply a subclass of `type` with a `__new__` method that creates the class object by running the default method in `type`. A metaclass `__new__` like this is run by the `__call__` method inherited from `type` by virtue of normal inheritance overrides; this method typically performs whatever augmentation is required and calls the `type` superclass's `__new__` method to create and return the new class object:

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        # Run by inherited type.__call__
        return type.__new__(meta, classname, supers, cl
```

This metaclass doesn't really do anything (we might as well let the default `type` create the class), but it demonstrates the way a metaclass taps into the metaclass hook to customize—because the metaclass is called at the end of a `class` statement, and because the `type` object's `__call__` dispatches to the `__new__` and `__init__` methods, code we provide in these methods can manage all the classes created from the metaclass.

To demo, Example 40-1 is our inane metaclass again, but in more tangible form, with prints added to the metaclass and the file at large to trace the process.

Example 40-1. metaclass1.py

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new:', meta, classname, super
        return type.__new__(meta, classname, supers, cl

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaOne):      # Inherits fr
    data = 1                               # Class data
    def meth(self, arg):                   # Class metho
        return self.data + arg

print('Making instance')
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

Here, class `Hack` inherits from `Super` and is an instance of `MetaOne`, but `X` is an instance of and inherits from `Hack`. When run, notice how the metaclass is invoked at the *end* of the `class` statement and before we ever make an instance of the new class—*metaclasses* process *classes*, and classes process *nonclass* instances:

```
$ python3 metaclass1.py
Making class
In MetaOne.new:
...<class '__main__.MetaOne'>
```

```
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <funct
Making instance
Attrs: 1 3
```

Presentation note: this chapter's examples often truncate hex addresses and omit some irrelevant built-in `__X__` names in namespace dictionaries, both for brevity and because built-in attributes tend to change over time. Run these examples on your own for full, if transient, fidelity.

Customizing Construction and Initialization

Metaclasses can also tap into the `__init__` protocol invoked by the type object's `__call__`. In general, `__new__` creates and returns the class object, and `__init__` initializes the already created class passed in as an argument. These methods work in *non-type* classes, too, but `__new__` is rare in such classes. Metaclasses can use either or both hooks to manage classes at creation time, as Example 40-2 illustrates.

Example 40-2. metaclass2.py

```
class MetaTwo(type):
    def __new__(meta, classname, supers, classdict):
        print()
        print('In MetaTwo.new:', meta, classname, super
        return type.__new__(meta, classname, supers, cl

    def __init__(Class, classname, supers, classdict):
        print()
        print('In MetaTwo.init:', Class, classname, sup
        print('...init class object:', list(Class.__dic

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaTwo):     # Inherits fr
    data = 1                              # Class data
    def meth(self, arg):                  # Class metho
       return self.data + arg

print('\nMaking instance')
```

```
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

In this case, the class *initialization* method is run after the class *construction* method, but both methods run at the end of the `class` statement and before any nonclass instances are made. Conversely, an `__init__` in `Hack` would run later at *nonclass-instance* creation time and would not be affected or run by the metaclass's `__init__`:

```
$ python3 metaclass2.py
Making class
In MetaTwo.new:
...<class '__main__.MetaTwo'>
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <funct

In MetaTwo.init:
...<class '__main__.Hack'>
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <funct
...init class object: ['__module__', 'data', 'meth', '_

Making instance
Attrs: 1 3
```

Other Metaclass Coding Techniques

Although redefining the `type` superclass's `__new__` and `__init__` methods is the most common way to insert logic into the class object creation process with the metaclass hook, other schemes are possible. They may not dovetail as neatly into the notion of metaclass methods we'll study ahead, but they do support creation-time tasks.

Using simple factory functions

For example, metaclasses need not really be classes at all. As we've learned, the `class` statement issues a simple call to create a class at the conclusion of its processing. Because of this, *any callable object* can, in principle, be used

as a metaclass, provided it accepts the arguments passed and returns an object compatible with the intended class. In fact, a simple object factory function may serve just as well as a `type` subclass, as Example 40-3 demonstrates.

Example 40-3. metaclass3.py

```
# A simple function can serve as a metaclass too

def MetaFunc(classname, supers, classdict):
    print('In MetaFunc:', classname, supers, classdict,
    return type(classname, supers, classdict)

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaFunc):           # Run
    data = 1                                     # Func
    def meth(self, arg):
        return self.data + arg

print('Making instance')
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

When run, the function is called at the end of the declaring `class` statement, and it returns the expected new class object. The function is simply catching the call that the `type` object's `__call__` normally intercepts by default:

```
$ python3 metaclass3.py
Making class
In MetaFunc:
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <funct
Making instance
Attrs: 1 3
```

Technically speaking, such a plain function used as a metaclass can return *anything*: whatever it returns is assigned to the new class's name, whether it's a `type` instance or not. Other kinds of results may not support later instance

creation, but this blurs the distinction between metaclasses and class decorators—both rebind names in the end.

Overloading class creation calls with normal classes

Because normal (a.k.a. *nonclass*) instances can respond to call operations with operator overloading, they can serve in some metaclass roles, too, much like the preceding function. The output of Example 40-4 is similar to the prior class-based versions, but it's based on a simple class—one that doesn't inherit from `type` at all and provides a `__call__` for its instances that catches the metaclass call using normal operator overloading.

All classes are created from a metaclass, even if it's the default `type` metaclass, but here we're using a *nonclass instance* of a class for the "metaclass." Note that `__new__` and `__init__` must use different names here, or else they will run when the `MetaObj` instance is *created*, not when that instance is later called in the role of metaclass after `Hack`'s `class` statement. The `__call__` here mimics part of what `type`'s method does.

Example 40-4. metaclass4.py

```
# A normal class instance can serve as a metaclass too

class MetaObj:
    def __call__(self, classname, supers, classdict):
        print('In MetaObj.call:', classname, supers, cl
        Class = self.__New__(classname, supers, classdi
        self.__Init__(Class, classname, supers, classdi
        return Class

    def __New__(self, classname, supers, classdict):
        print('In MetaObj.new: ', classname, supers, cl
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classc
        print('In MetaObj.init:', classname, supers, cl
        print('...init class object:', list(Class.__dic

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaObj()):         # MetaC
```

```
        data = 1                                    # Calle
        def meth(self, arg):
            return self.data + arg

    print('Making instance')
    X = Hack()
    print('Attrs:', X.data, X.meth(2))
```

When run, the three methods are dispatched via the normal instance's `__call__` inherited from its normal class, but without any dependence on `type` dispatch mechanics or semantics. This routing is largely about `class` alone:

```
    $ python3 metaclass4.py
    Making class
    In MetaObj.call:
    ...Hack
    ...(<class '__main__.Super'>,)
    ...{'__module__': '__main__', 'data': 1, 'meth': <funct
    In MetaObj.new:
    ...Hack
    ...(<class '__main__.Super'>,)
    ...{'__module__': '__main__', 'data': 1, 'meth': <funct
    In MetaObj.init:
    ...Hack
    ...(<class '__main__.Super'>,)
    ...{'__module__': '__main__', 'data': 1, 'meth': <funct
    ...init class object: ['__module__', 'data', 'meth', '_
    Making instance
    Attrs: 1 3
```

In fact, we can use normal superclass inheritance to acquire the call interceptor in this coding model—the superclass in Example 40-5 serves essentially the same role as `type`, at least in terms of metaclass dispatch. Such code may be atypical, but it demos the underlying metaclass dispatch model.

Example 40-5. metaclass5.py

```
    # Instances inherit from classes and their supers normc

    class SuperMetaObj:
```

```
    def __call__(self, classname, supers, classdict):
        print('In SuperMetaObj.call:', classname, super
        Class = self.__New__(classname, supers, classdi
        self.__Init__(Class, classname, supers, classdi
        return Class

class SubMetaObj(SuperMetaObj):
    def __New__(self, classname, supers, classdict):
        print('In SubMetaObj.new: ', classname, supers,
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classc
        print('In SubMetaObj.init:', classname, supers,
        print('...init class object:', list(Class.__dic

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=SubMetaObj()):   # Invoke S
   …rest of file same as Example 40-4…
```

This example's output is largely the same as that of its predecessor but reflects normal inheritance at work:

```
$ python3 metaclass5.py
Making class
In SuperMetaObj.call:
…as before…
In SubMetaObj.new:
…as before…
In SubMetaObj.init:
…as before…
Making instance
Attrs: 1 3
```

Although such alternative forms work, most metaclasses achieve their creation-time goals by redefining the `type` superclass's `__new__` and `__init__`; in practice, this may be simpler than other schemes. Regardless of its coding, though, this metaclass role broadly intersects with class decorators—as the next section will demonstrate.

Managing Classes with Metaclasses and Decorators

Now that we understand the hook metaclasses use to insert code to be run at class construction time, let's put it to better use. In general, such code can be used to augment classes arbitrarily, and in many of the same ways as the *class decorators* of Chapter 39. This doesn't make these two tools identical—metaclasses also support the notion of inherited methods coming up later—but they are functionally redundant in some roles.

Adding methods to classes

To demo both this equivalence and more realistic usage, Example 40-6 uses metaclasses to augment the set of methods available in classes by *inserting* new methods at class-creation time as sketched in the abstract earlier—not the sort of thing most programmers do on a day-to-day basis, but potentially useful in tools and libraries nonetheless.

Example 40-6. extend_meta.py

```
"Extend a class with a metaclass"

def triple(obj):
    return obj.value * 3

def concat(obj):
    return obj.value + 'Code!'

class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['triple'] = triple
        classdict['concat'] = concat
        return type.__new__(meta, classname, supers, cl

class Client1(metaclass=Extender):
    def __init__(self, value):
        self.value = value
    def double(self):
        return self.value * 2

class Client2(metaclass=Extender):
    value = 'grok'

X = Client1('hack')
```

```
print(X.double(), X.triple(), X.concat(), sep='\n')

Y = Client2()
print(Y.triple(), Y.concat(), sep='\n')
```

Recall again that class methods are simply *functions* that normally receive an instance through which they are called. The metaclass in this code leverages this to insert two functions into each of its client classes when those classes are made. The net effect provides methods and behavior for clients that do not exist in their `class` statements:

```
$ python3 extend_meta.py
hackhack
hackhackhack
hackCode!
grokgrokgrok
grokCode!
```

Of course, the methods inserted here might be coded in a *superclass* and inherited by clients as usual, but the metaclass here is free to select inserted methods based on conditions tested whenever clients are built. As covered ahead, we can also *almost* do the same with metaclass *methods*, but these methods are inherited only by classes and wouldn't work in this example; the methods inserted here are instead available to later class instances like `X` and `Y`.

And while this may seem novel, it's easy to accomplish the same result with class decorators. As we've learned, there are indeed some striking similarities between these tools:

- *Class decorators* work by rebinding class names to the result of a callable at the end of a `class` statement after the new class has been created.
- *Metaclasses* work by routing class-object creation through a callable at the end of a `class` statement in order to create the new class.

The sum makes these two tools functionally equivalent, at least in terms of class-creation dispatch. Example 40-7 illustrates this equivalence by recoding the same augmentation with a class decorator instead of a metaclass.

Example 40-7. extend_deco.py

```
"Extend a class with a decorator"

def triple(obj):
    return obj.value * 3

def concat(obj):
    return obj.value + 'Code!'

def extender(aClass):
    aClass.triple = triple                  # Manages
    aClass.concat = concat                  # Same as
    return aClass

@extender
class Client1:                              # Client1
    def __init__(self, value):              # Rebound
        self.value = value
    def double(self):
        return self.value * 2

@extender
class Client2:
    value = 'grok'

X = Client1('hack')
print(X.double(), X.triple(), X.concat(), sep='\n')

Y = Client2()
print(Y.triple(), Y.concat(), sep='\n')
```

When run, this class-decorator version's output is identical to that of the metaclass variant in Example 40-6. It works the same because both decorator and metaclass are called at the end of a `class` statement and return an object to which the class's name is assigned. Decorators may be closest to the metaclass `__call__` because they are called to return an object, but like metaclass `__init__`, the class has already been built by the time a decorator runs. Metaclass `__call__` runs `__new__` and `__init__`, and metaclasses can augment in any of these three methods.

Automatically decorating class methods

Perhaps more interesting, metaclasses and decorators can both augment individual methods of classes. To demo, we'll use the utility module in Example 40-8, which resurrects the tracer and timer function decorators we coded in the prior chapter. This module is entirely review, so we'll defer to Chapter 39 for more details (and promise that this is the last mileage we'll get from this code).

Example 40-8. decorators.py

```
import time

def tracer(func):                           # Use functic
    calls = 0                               # Else self i
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print(f'call {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return onCall

def timer(label='', trace=True):                # On de
    def onDecorator(func):                      # On @:
        def onCall(*args, **kargs):             # On cc
            start   = time.perf_counter()       # State
            result  = func(*args, **kargs)
            elapsed = time.perf_counter() - start
            onCall.alltime += elapsed
            if trace:
                funcname, alltime = func.__name__, onCa
                print(f'{label}{funcname}: {elapsed:.5f
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator
```

As we learned in Chapter 39, to use these decorators manually, we simply import them from the module and decorate each function we wish to augment. While this suffices for one-off augmentations, it requires us to add decoration syntax before *each* method we wish to trace or time and to later remove that syntax when we no longer desire the extensions (or use the `-O` trick we'll

skip here). If we want to trace or time *every* method of a class, this can become tedious—and may not be possible at all in more dynamic contexts that depend upon runtime parameters.

To do better, Example 40-9 uses a metaclass to add a decorator to *each* of a class's methods automatically. Because the metaclass controls decoration, it can predicate decoration on runtime checks. As a bonus, it can be used for *any* decoration: the decorator to apply to methods is passed as a top-level argument, and hence is allowed to vary per class.

Example 40-9. decoall_meta.py

```
"Apply any decorator to all methods of a class, with a

from types import FunctionType
from decorators import tracer, timer

def decorateAll(decorator):
    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict)
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval
            return type.__new__(meta, classname, supers
    return MetaDecorate

class Person(metaclass=decorateAll(tracer)):        # Us
    def __init__(self, name, pay):                  # Pc
        self.name = name
        self.pay  = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

def tester(aPerson):
    sue = aPerson('Sue Jones', 100_000)
    bob = aPerson('Bob Smith', 50_000)
    print(f'{sue.name=}, {bob.name=}')
    sue.giveRaise(.10)
    print(f'{sue.pay=:,.2f}')
    print('Last names:', sue.lastName(), bob.lastName()
```

```
if __name__ == '__main__': tester(Person)
```

When this code is run as is, its output traces calls to every method of the client class because every method has been automatically decorated by the metaclass:

```
$ python3 decoall_meta.py
call 1 to __init__
call 2 to __init__
sue.name='Sue Jones', bob.name='Bob Smith'
call 1 to giveRaise
sue.pay=110,000.00
call 1 to lastName
call 2 to lastName
Last names: Jones Smith
```

Really, this result reflects a *combination* of decorator and metaclass—the metaclass automatically applies the function decorator to every method at class creation time, and the function decorator automatically intercepts method calls in order to print the trace messages in this output. The combo "just works," thanks to the generality of both tools.

To apply a *different* decorator to the methods, simply replace the decorator name in the `class` header line. To use the timer function decorator shown earlier, for example, use either of the last two header lines in the following for our `Person` class—the first accepts the timer's default arguments, and the second specifies label text (though methods may run too fast to register runtimes as is: add `time.sleep(`*`seconds`*`)` calls to pause for a better time):

```
class Person(metaclass=decorateAll(tracer)):
class Person(metaclass=decorateAll(timer())):
class Person(metaclass=decorateAll(timer(label='**'))):
```

And as you might expect by now, class decorators intersect with metaclasses here, too. Example 40-10 replaces the preceding example's metaclass with a class decorator. Really, it uses a class decorator that applies a function

decorator to each method of a decorated class. Python's decorators naturally support arbitrary nesting and combinations.

Example 40-10. decoall_deco.py

```
"Apply any decorator to all methods of a class, with a

from types import FunctionType
from decoall_meta import tester
from decorators import tracer, timer

def decorateAll(decorator):
    def DecoDecorate(aClass):
        for attr, attrval in aClass.__dict__.items():
            if type(attrval) is FunctionType:
                setattr(aClass, attr, decorator(attrval
        return aClass
    return DecoDecorate

@decorateAll(tracer)                          # Use cla
class Person:                                 # Applies
    def __init__(self, name, pay):            # Person
        self.name = name                      # Person
        self.pay  = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

if __name__ == '__main__': tester(Person)
```

When this code is run, the class decorator applies the tracer function decorator to every method, which in turn produces a trace message on calls (the output is the same as that of the preceding metaclass version of this example):

```
$ python3 decoall_deco.py
…same as Example 40-9…
```

Much as before, we simply mod the `@` decorator line to apply a different decorator or provide different arguments for it. Test the following on your own to see the effect (and again add sleep calls as needed to boost times):

```
@decorateAll(tracer)                    # Apply trc
@decorateAll(timer())                   # Apply tim
@decorateAll(timer(label='**'))         # Decorator
@decorateAll(timer(label='**', trace=0))    # More decc
```

Notice that this class decorator returns the augmented class, not a proxy wrapper. As for the metaclass version, this retains the type of the original class—an instance of `Person` is still an instance of `Person`. This may matter when type testing is used. The class's *methods* are not their original functions because they are rebound to decorators, but this is likely less important in practice, and it's true in the metaclass alternative as well.

So far, what we've seen of metaclasses makes them seem largely redundant with decorators—but we have not yet seen all there is to see. As teased earlier, metaclasses may also provide behavior to their instance classes by defining *methods*, which have no direct counterpart in decorators. These methods, however, come with a twist that limits their scope. To understand both metaclasses' methods and their limiting twist, though, we first have to factor metaclasses into Python attribute resolution, a.k.a. *inheritance*, at large. The next section takes us down this prerequisite path.

Inheritance: The Finale

Because metaclasses are coded in similar ways to inheritance superclasses, their scope can be confusing at first glance. In short, there are really two class trees searched by Python inheritance—a *primary* tree formed by a class and that class's superclasses, along with a *secondary* tree formed by a class's metaclass and that metaclass's superclasses. The secondary tree is also called the "type" tree because it stems from `type`. In more detail, here is how this pans out:

Metaclasses inherit from the `type` class (usually)

Although they have a special role, metaclasses are coded with normal `class` statements and follow the usual OOP model in Python. As subclasses of `type`, they can redefine the `type` object's methods to customizing classes as needed. Per the prior section, metaclasses often redefine the `type` class's `__new__` and `__init__` to intercept class creation and initialization. Metaclasses can also define methods

for their instance classes and may be simple functions or other callables that return arbitrary objects.

Metaclass attributes **are not** *acquired by class instances*

Metaclass declarations specify an *instance* relationship, which is not quite the same as superclass inheritance. Behavior defined in a metaclass applies to the classes made from it but *not* to these classes' own *nonclass* instances. Inheritance for a nonclass instance searches only the instance and the *primary* tree formed by its class and that class's superclasses; the secondary metaclass tree is never included in this search. Hence, nonclass instances get behavior from classes and superclasses but not from metaclasses.

Metaclass attributes **are** *acquired by classes as a fallback*

By contrast, classes *do* acquire methods of their metaclasses by virtue of the *instance* relationship. Metaclasses define a separate inheritance tree, which is a source of class behavior that processes classes themselves. For classes only, inheritance first searches the *primary* tree formed by the class and its superclasses and then falls back on the *secondary* tree formed by the class's metaclass and *its* superclasses as a separate search. When a name is available to a class in *both* a metaclass and a superclass, the superclass version is used.

Metaclass declarations are also inherited by subclasses

The `metaclass=M` declaration in a user-defined class is also *inherited* by the class's normal subclasses in much the same way that superclass links are inherited by subclasses. Thus, the metaclass will run for the construction of each class that inherits this specification in a superclass inheritance chain.

This model's conflation of classes and metaclasses can make terminology challenging, but it may be easier to understand in code than in prose. To illustrate the preceding points, consider the code in Example 40-11.

Example 40-11. metainstance.py

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        print('In Meta.new:', classname)
        return type.__new__(meta, classname, supers, cl
    def meth3(self):
        return 'three!'
```

```
class Super(metaclass=Meta):           # Metaclass inhe
    def meth2(self):                   # Meta run twice
        return 'two!'

class Sub(Super):                      # Superclass: ir
    def meth1(self):                   # Classes inheri
        return 'one!'                  # But not from m
```

When this file's code is run (as a script or module), the metaclass handles construction of *both* client classes. When those classes are later used, nonclass *instances* inherit class attributes but *not* metaclass attributes:

```
>>> from metainstance import *         # Runs class sta
In Meta.new: Super
In Meta.new: Sub

>>> X = Sub()                 # Nonclass instance of user-
>>> X.meth1()                 # Inherited from Sub
'one!'
>>> X.meth2()                 # Inherited from Super
'two!'
>>> X.meth3()                 # Not inherited from metacla
AttributeError: 'Sub' object has no attribute 'meth3'.
```

By contrast, *classes* both inherit names from their superclasses and acquire names from their metaclass—whose linkage in this example is *itself* inherited from a superclass by `Sub`:

```
>>> Sub.meth1(X)              # Own method
'one!'
>>> Sub.meth2(X)              # Inherited from Super
'two!'
>>> Sub.meth3()               # Acquired from metaclass
'three!'
>>> Sub.meth3(X)              # Not a normal instance meth
TypeError: Meta.meth3() takes 1 positional argument but
```

Notice how the last of the preceding calls fails when we pass in an instance because the name resolves to a *metaclass method*, not a normal instance method. In fact, both the *source* of a name and the *object* through which you

fetch it matter here. Methods acquired from metaclasses are bound to the subject *class*, while methods from normal classes are plain *functions* when fetched through the class but bound with an *instance* when fetched through the instance:

```
>>> Sub.meth3
<bound method Meta.meth3 of <class 'metainstance.Sub'>>
>>> Sub.meth2
<function Super.meth2 at 0x1085f7d80>
>>> X.meth2
<bound method Super.meth2 of <metainstance.Sub object a
```

We studied the last two of these cases before in Chapter 31's bound-method coverage. The first case is reminiscent of Chapter 32's *class methods* but is technically new here. We'll explore class methods in more detail later. First, though, to understand why metaclass methods aren't available to normal instances, we need to clarify the metaclass/superclass distinction further.

Metaclass Versus Superclass

In even simpler terms, watch what happens in the following: as an *instance* of the `M` metaclass type, class `C` acquires `M`'s attribute, but this attribute is not made available for inheritance by `C`'s own instance `I`—the acquisition of names by metaclass instances is *distinct* from the normal inheritance used for nonclass instances:

```
>>> class M(type): attr = 1
>>> class C(metaclass=M): pass          # C is meta inst
>>> I = C()                             # I inherits frc
>>> C.attr
1
>>> I.attr
AttributeError: 'C' object has no attribute 'attr'
>>> 'attr' in C.__dict__, 'attr' in M.__dict__
(False, True)
```

By contrast, if `M` morphs from metaclass to superclass, then names in superclass `S` become available to later instances of `C` by *inheritance*—that is, by searching the `__dict__` attribute dictionaries of objects in the MRO

of `I`'s class as usual, much like the `mapattrs` example we coded back in "Example: Mapping Attributes to Inheritance Sources" (a nonclass-instance-only tool):

```
>>> class S: attr = 1
>>> class C(S): pass                    # C is type inst
>>> I = C()                             # I inherits fro
>>> C.attr
1
>>> I.attr
1
>>> 'attr' in C.__dict__, 'attr' in S.__dict__
(False, True)
```

This is why metaclasses often do their work by manipulating a new class's namespace dictionary if they wish to influence the behavior of later instance objects—instances will see names in their *class* but not its *metaclass*. Watch what happens, though, if the same name is available in *both* attribute sources—the *inheritance* name in the primary superclass tree is used instead of the *instance* acquisition name in the secondary metaclass tree:

```
>>> class M(type): attr = 1
>>> class S: attr = 2
>>> class C(S, metaclass=M): pass       # Supers have p
>>> I = C()                             # Classes searc
>>> C.attr, I.attr                      # Instances sea
(2, 2)
>>> 'attr' in C.__dict__, 'attr' in S.__dict__, 'attr'
(False, True, True)
```

This is true regardless of the relative height of the inheritance and instance sources—superclass inheritance always beats metaclass acquisition because primary trees are searched before secondary trees:

```
>>> class M(type): attr = 1
>>> class S2: attr = 2
>>> class S1(S2): pass
>>> class C(S1, metaclass=M): pass      # Super two lev
>>> I = C()
```

```
>>> C.attr, I.attr
(2, 2)
```

In fact, classes acquire metaclass attributes through their `__class__` link, in the same way that nonclass instances inherit from classes through their `__class__` —which makes sense, given that classes are also instances of metaclasses. The chief distinction is that instance inheritance does not also follow a class's `__class__` but instead restricts its scope to the `__dict__` of each class in the superclass tree, following `__bases__` along the way:

```
>>> I.__class__                 # Followed by inheritanc
<class '__main__.C'>
>>> C.__bases__                 # Followed by inheritanc
(<class '__main__.S1'>,)
>>> C.__class__                 # Followed by instance c
<class '__main__.M'>
```

Really, though, Python checks the `__dict__` of each class on the class's *MRO* before falling back on doing the same for its metaclass—and runs the second of these steps only for fetches run on classes, not nonclass instances:

```
>>> [x.__name__ for x in C.__mro__]     # See Chapter 3
['C', 'S1', 'S2', 'object']
>>> [x.__name__ for x in M.__mro__]     # Primary/secor
['M', 'type', 'object']
```

These two MROs for class and metaclass are simply the flattened versions of what we called the *primary* and *secondary* class trees earlier. Nonclass-instance inheritance searches just the first, but class inheritance searches both if needed. In fact, metaclasses work in much the same way, as the next section explains.

Metaclass Inheritance

As it turns out, instance inheritance works in similar ways, whether the "instance" is a nonclass instance created from a class or a class created from a metaclass derived from `type` . While classes straddle the primary and secondary trees, both trees use the same mechanism to look up names. This

yields a single attribute search procedure spanning two trees, which fosters the parallel notion of *metaclass* inheritance hierarchies.

The following demos this conceptual merger. In it, instance `I` inherits from all its classes; class `C` inherits from both superclasses and metaclasses; and metaclass `M1` inherits from higher metaclasses:

```
>>> class M2(type): attr4 = 4                  # Metacla
>>> class M1(M2):   attr3 = 3                  # Gets __

>>> class S: attr2 = 2                         # Supercl
>>> class C(S, metaclass=M1): attr1 = 1        # Gets __

>>> I = C()                                    # I gets
>>> I.attr1, I.attr2                           # Instanc
(1, 2)
>>> C.attr1, C.attr2, C.attr3, C.attr4         # Class g
(1, 2, 3, 4)
>>> M1.attr3, M1.attr4                         # Metacla
(3, 4)
>>> I.attr3
AttributeError: 'C' object has no attribute 'attr3'. Di
```

The failure at the end of this listing is pivotal. Both inheritance paths—class and metaclass—employ the same links to build class MROs scanned by inheritance. But this is not applied *transitively*—instances do not inherit their class's metaclass names, though they may request them explicitly:

```
>>> I.__class__                # Links used at instance
<class '__main__.C'>
>>> C.__bases__
(<class '__main__.S'>,)

>>> C.__class__                # Links used at class aj
<class '__main__.M1'>
>>> M1.__bases__               # Link also used in meta
(<class '__main__.M2'>,)

>>> I.__class__.attr4          # Route inheritance to t
4
```

```
>>> I.attr4                    # Though class's __class
AttributeError: 'C' object has no attribute 'attr4'. Di
```

And as usual, both trees have flattened MROs and instance links actually used by inheritance:

```
>>> M1.__class__                          # Metaclasses c
<class 'type'>
>>> [x.__name__ for x in C.__mro__]       # __bases__ tre
['C', 'S', 'object']
>>> [x.__name__ for x in M1.__mro__]      # __bases__ tre
['M1', 'M2', 'type', 'object']
```

If you care about metaclasses—or must use code that does—study these examples carefully. In effect, nonclass instances have no `__bases__` to search but follow `__class__` to `__bases__` once, and classes follow `__bases__` before following a single `__class__` to another `__bases__`. Because this is crucial to the meaning of attribute names in Python, the next section begins making it more formal.

Python Inheritance Algorithm: The Simple Version

Now that we know about metaclass acquisition, we're finally able to formalize the inheritance rules that they augment. Inheritance deploys two distinct but similar lookup routines, both of which are founded on MROs computed from the prior section's `__bases__` links per Chapter 32. The combination yields the following first-cut definition of Python's attribute inheritance algorithm run for explicit attribute fetches.

To look up an attribute name:

1. From a nonclass *instance* I, search the instance, then its class, and then all its superclasses, using:
   a. The `__dict__` of the instance I
   b. The `__dict__` of all classes on the `__mro__` found at I's `__class__`, from left to right
2. From a *class* C, search the class, then all its superclasses, and then its metaclasses tree, using:

a. The `__dict__` of all classes on the `__mro__` found at C itself, from left to right

b. The `__dict__` of all metaclasses on the `__mro__` found at C's `__class__`, from left to right

3. In rules *1* and *2*, also allow for these exceptions covered ahead:

– Give precedence to *data descriptors* present in step *b* sources

– Skip step *a* and begin the search at step *b* for *built-in operations*

– Perform a custom MRO search for a proxied object in `super` objects

Rules *1* and *2* are applied for normal, explicit attribute fetch only, and there are exceptions for built-ins, descriptors, and `super`, which we'll clarify in a moment. In addition, a `__getattr__` or `__getattribute__` may also be used for missing or all names, respectively, per Chapter 38, and attribute assignment follows different rules.

The first two rules here, however, are the essentials. They specify the inheritance search of *two* separate trees, which works the same in each tree but spans trees for *class* inheritance alone. The MRO pseudocode of Chapter 32 summarizes this even more concisely:

```
[I.__dict__] + [x.__dict__ for x in I.__class__.__mro__

[x.__dict__ for x in C.__mro__] + [x.__dict__ for x in
```

In this, the first line is sources searched by rule *1* for inheritance from a nonclass instance `I`—both the instance itself and the flattened version of the *primary* class tree. The second line is rule *2*'s sources for inheritance from a class `C`—the flattened versions of both the *primary* class tree and the *secondary* metaclass tree.

Put another way, nonclass instances and classes both search a class tree's MRO at their `__class__` as a second step, but classes first search their own MRO's superclass tree instead of a single `__dict__` namespace dictionary. In code, `class` header lines define both of the trees searched by listing superclasses and specify links to secondary trees with `metaclass` keywords. When omitted, superclass defaults to `object` and metaclass to `type`.

Most programmers need only be aware of the first of these rules (*1*) and perhaps the first step of the second (*2a*). There's an extra acquisition step added for metaclasses (*2b*), but it's essentially the same as others—a subtle equivalence, to be sure, but metaclass acquisition is not as novel as it may seem. It's just one step of the procedure.

The descriptors deviation

At least, that's the *normal*—and sugarcoated—case. The prior section listed its exceptions separately because they don't matter in most code and complicate the algorithm substantially. First among these, inheritance also has a special-case coupling with Chapter 38's attribute descriptors. In short, *data descriptors*—those that define `__set__` methods to intercept assignments—are given precedence, such that their names override other inheritance sources.

This exception ostensibly serves some practical roles. For example, it is used to ensure that the special `__class__` and `__dict__` attributes cannot be redefined by the same names in an instance's own `__dict__`. In the following, these data-descriptor names in the class *override* same names created in the instance's attribute dictionary:

```
>>> class C: pass                          # Inheritanc
>>> I = C()                                # Class data
>>> I.__class__, I.__dict__
(<class '__main__.C'>, {})

>>> I.__dict__['book'] = 'lp6e'            # Dynamic da
>>> I.__dict__['__class__'] = 'hack'       # Assign key
>>> I.__dict__['__dict__']  = {}

>>> I.book                                 # I.name com
'lp6e'                                     # But I.__cl
>>> I.__class__, I.__dict__
(<class '__main__.C'>, {'book': 'lp6e', '__class__': 'h
```

This data-descriptor exception is tested *before* the preceding two inheritance rules as a preliminary step, may be more important to Python implementers than Python programmers, and can be reasonably ignored by most application code in any event—that is, unless *you* code data descriptors of your own, which follow the same inheritance special case:

```
>>> class D:
        def __get__(self, instance, owner): print('D.__
        def __set__(self, instance, value): print('D.__

>>> class C: d = D()            # Data-descriptor attri
>>> I = C()
>>> I.d                         # Inherited data descri
D.__get__
>>> I.d = 1
D.__set__
>>> I.__dict__['d'] = 'hack'    # Define same name in i
>>> I.d                         # But doesn't hide data
D.__get__
```

Conversely, if this descriptor did *not* define a `__set__`, the name in the instance's dictionary *would* hide the name in its class instead, per normal inheritance:

```
>>> class D:
        def __get__(self, instance, owner): print('D.__

>>> class C: d = D()
>>> I = C()
>>> I.d                         # Inherited nondata des
D.__get__
>>> I.__dict__['d'] = 'hack'    # Hides class names per
>>> I.d
'hack'
```

In both cases, Python automatically runs the descriptor's `__get__` when it's found by inheritance rather than returning the descriptor object itself—part of the implicit attribute magic we met earlier in the book. The special status afforded to data descriptors, however, also modifies the meaning of attribute *inheritance* and, thus, the meaning of names in your code. The next section's final swing at a formal inheritance algorithm takes this into account.

Python Inheritance Algorithm: The Less Simple Version

With both the data descriptor special case and general descriptor invocation factored in with class and metaclass trees, Python's formal inheritance algorithm can be stated as follows—a complex procedure which assumes knowledge of descriptors, metaclasses, and MROs but is the final arbiter of attribute names nonetheless.

To look up an attribute name:

1. From a nonclass *instance* I, search the instance, its class, and its superclasses, as follows:
   a. Search the `__dict__` of all classes on the `__mro__` found at I's `__class__`
   b. If a data descriptor was found in step *a*, call its `__get__` and exit
   c. Else, return a value in the `__dict__` of the instance I
   d. Else, call a nondata descriptor or return a value found in step *a*
2. From a *class* C, search the class, its superclasses, and its metaclasses tree as follows:
   a. Search the `__dict__` of all metaclasses on the `__mro__` found at C's `__class__`
   b. If a data descriptor was found in step *a*, call its `__get__` and exit
   c. Else, call any descriptor or return a value in the `__dict__` of a class on C's own `__mro__`
   d. Else, call a nondata descriptor or return a value found in step *a*
3. In rules *1* and *2*, *built-in* operations essentially use just step *a* sources (see ahead)
4. The `super` built-in performs a custom MRO search for a proxied object (see ahead)

Some fine print here: in this procedure, items are attempted in sequence as numbered, and Python runs at most *one* (for instances) or *two* (for classes) MRO searches per name lookup—the first appearance of a name in a given MRO wins, regardless of its kind. Because each MRO (a.k.a. `__mro__`) is a flattened representation of a class tree with duplicates removed, you can also think of these as one or two *tree* searches.

In addition, the implied `object` superclass provides some defaults at the top of every class and metaclass tree (that is, at the end of every MRO). And

beyond all this, method `__getattr__` may be run if defined when an attribute is not found, and method `__getattribute__` may be run for every attribute fetch, though they are special-case extensions to the name lookup model (really, the latter replaces inheritance for the defining class's instances, and can trigger the former with an attribute exception). See Chapter 38 for more on these tools as well as descriptors.

Also, note here again that this algorithm's first two steps apply only to *normal* and *explicit* attribute *fetch*. The rules for attribute *assignment* vary; the *implicit* lookup of method names for *built-ins* doesn't follow these rules in full; and the *proxied* lookup of attributes performed by `super` is entirely custom. The next sections cover these exceptions separately.

The assignment addendum

The prior section defines inheritance in terms of attribute *reference* (a.k.a. fetch or lookup), but parts of it apply to attribute *assignment* as well. As we've learned, assignment normally changes attributes in the *subject* object itself, but inheritance is also invoked on assignment to test first for some of Chapter 38's attribute management tools, including descriptors, properties, and `__setattr__`. When present, such tools intercept attribute assignment and may implement it arbitrarily.

For example, attribute assignment always invokes a `__setattr__` if present, much as a `__getattribute__` is run for all references. Moreover, a *data descriptor* with a `__set__` method is acquired from a class by inheritance using the MRO and has precedence over the normal storage model. In terms of the prior section's rules:

- When applied to an *instance*, attribute assignments essentially follow steps *a* through *c* of rule *1*, searching the instance's superclass tree, though step *b* calls `__set__` instead of `__get__`, and step *c* stops and stores in the instance instead of attempting a fetch.
- When applied to a *class*, attribute assignments run the same procedure on the class's metaclass tree: roughly the same as rule *2*, but step *c* stops and stores in the class.

Because descriptors are also the basis for other advanced attribute tools such as *properties* and *slots*, this inheritance precheck on assignment is utilized in

multiple contexts. The net effect is that descriptors are treated as an inheritance special case for *both* reference and assignment.

The super supplement

Even for attribute *reference*, there are two special cases that are exempt from inheritance's normal rules. For one, the `super` built-in function we studied in Chapter 32 does not use normal inheritance.

As we learned, for the proxy objects returned by `super`, attributes are resolved by a special context-sensitive scan of a limited *tail* portion of a *different* object's MRO. This custom scan searches the MRO of an implicit instance's class, choosing the first descriptor or value found in a class *following* the class containing the `super` call. This scan is used *instead* of running full inheritance—which is used on the `super` object itself only if this special-case scan fails.

See Chapter 32 for more coverage of `super`. While this built-in may be convenient in some simple roles, it comes with substantial complexity in others and adds a convoluting footnote to inheritance itself.

The built-ins bifurcation

As we've also learned, other *built-ins* don't follow inheritance's normal rules either. Instances and classes may both be skipped for the *implicit* method-name fetches of built-in operations, as a special case that differs from *explicit* name references. Because this is a *context-specific* divergence, it's easier to demonstrate in code than to weave it into a single algorithm. In the following, `str` is the built-in, `__str__` is its explicit-name equivalent, and the *nonclass* instance is inconsistently skipped by the built-in only:

```
>>> class C:                                # Inheritance
        attr = 1                            # Built-ins s
        def __str__(self): return('class')

>>> I = C()
>>> I.__str__(), str(I)                     # Both from c
('class', 'class')

>>> I.__str__ = lambda: 'instance'
>>> I.__str__(), str(I)                     # Explicit=>i
```

```
('instance', 'class')

>>> I.attr                                  # Asymmetric
1
>>> I.attr = 2; I.attr
2
```

As you may expect by now, the same holds true for *classes*—explicit names start at the class, but built-ins start at the class's class—which is its *metaclass*, and defaults to `type`, which provides access to an implicit default:

```
>>> class D(type):
        def __str__(self): return('D class')
>>> class C(D):
        pass
>>> C.__str__(C), str(C)                    # Explicit=>s
('D class', "<class '__main__.C'>")

>>> class C(D):
        def __str__(self): return('C class')
>>> C.__str__(C), str(C)                    # Explicit=>c
('C class', "<class '__main__.C'>")

>>> class C(metaclass=D):
        def __str__(self): return('C class')
>>> C.__str__(C), str(C)                    # Built-in=>u
('C class', 'D class')
```

In fact, it can sometimes be nontrivial to know *where* a name comes from in this model since all classes in both trees also inherit from `object`—including the default `type` metaclass. In the following's explicit call, `C` gets a default `__str__` from `object` instead of the metaclass, per the first source of class inheritance (the class's own MRO, which is the primary tree); by contrast, the `str` built-in skips ahead to the metaclass as before:

```
>>> class C(metaclass=D):
        pass
>>> C.__str__(C), str(C)                    # Explicit=>c
("<class '__main__.C'>", 'D class')

>>> C.__str__
<slot wrapper '__str__' of 'object' objects>
```

```
>>> for k in (C, C.__class__, type):
        print([x.__name__ for x in k.__mro__])

['C', 'object']
['D', 'type', 'object']
['type', 'object']
```

This is why we've gone to such great lengths to root out the full specs of inheritance. While some code may never need to care about all its many plot twists, attribute inheritance is clearly a convoluted business in Python, and uncertainty is not generally compatible with engineering.

The Inheritance Wrap-Up

And with all those details in hand, you finally have the complete Python inheritance story—or at least as much as we can cover in this text. It's a tangled tale that today spans instances, classes, superclasses, metaclasses, descriptors, `super`, built-ins, and MROs, and all for the sake of looking up a simple attribute name.

Some practical needs warrant exceptions, of course, but you should carefully consider the implications of an object-oriented language that applies inheritance—its *foundational* operation—in such a *labyrinthian* fashion. At a minimum, this should underscore the importance of keeping your own code simple to avoid making it dependent on the darker corners of such convoluted rules. As always, your code's users and maintainers will be glad you did.

For more fidelity on this story, see Python's internal implementation of inheritance—a low-level but complete saga chronicled today in its files *object.c* and *typeobject.c*, the former for normal instances and the latter for classes. Delving into internals shouldn't be required to use Python, but it's the ultimate and sometimes sole source of truth in a complex and perpetually changing system. This is especially true in boundary cases born of accrued exceptions that raise the bar for learners and users, a downside we'll revisit briefly in the next and closing chapter.

For now, let's move on to one last bit of metaclass "magic"—its methods, which rely on its inheritance offshoot.

Metaclass Methods

Now that we have a handle on the way that metaclasses modify the inheritance of names, we can finally turn to their methods with full clarity. In short, *methods* in metaclasses are inherited by and process their instance *classes*—instead of the nonclass instances that classes make.

This makes metaclass methods similar in form and function to the *class methods* we studied in Chapter 32, though their class-focused behavior is automatic. Moreover, they are available only to classes due to the last section's inheritance rules—the limiting "twist" alluded to earlier. Metaclass methods also have no direct analog in class decorators, though decorators' freedom to return arbitrary objects makes nearly anything possible with imagination.

To demo the basics, the following's metaclass defines a method made available to its class's instances by metaclass acquisition (i.e., by inheritance from the secondary metaclass tree):

```
>>> class M(type):
        def z(cls):  print('M.z', cls)          # A met
        def y(cls):  print('M.y', cls)          # y is

>>> class C(metaclass=M):
        def y(self): print('C.y', self)         # A sim
        def x(self): print('C.x', self)         # Names
```

Methods fetched from the class are plain functions as usual, but those from a metaclass are automatically bound to the class from which they were fetched, as we saw in an earlier example:

```
>>> C.x
<function C.x at 0x10eaf4b80>
>>> C.y                                         # x and
<function C.y at 0x10eaf4ae0>
>>> C.z                                         # z acc
<bound method M.z of <class '__main__.C'>>
>>> C.z()                                       # Metac
M.z <class '__main__.C'>
```

As we've also seen, methods fetched through a nonclass instance are bound to the instance unless `classmethod` or `staticmethod` are used, though such instances are moot here because they do not inherit metaclass names:

```
>>> I = C()
>>> I.x()                                   # Insta
C.x <__main__.C object at 0x10e9d1400>
>>> I.y()
C.y <__main__.C object at 0x10e9d1400>
>>> I.z()                                   # Insta
AttributeError: 'C' object has no attribute 'z'
>>> I.x
<bound method C.x of <__main__.C object at 0x10e9d1400>
```

The only real new things about metaclass methods is that they are inherited only by classes and are bound to a class automatically. The latter half of this is explored in the next section.

Metaclass Methods Versus Class Methods

Though they differ in inheritance visibility, metaclass methods are designed to manage class-level data, just like the *class methods* we studied in Chapter 32. In fact, their roles can overlap—much as metaclasses can in general with class decorators—but metaclass methods are not accessible except through the class and do not require an explicit `classmethod` class-level declaration in order to be bound with the class.

In other words, metaclass methods can be thought of as *implicit* class methods, with limited *visibility*:

```
>>> class M(type):
        def a(cls):                          # Metaclass
            cls.x = cls.y + cls.z

>>> class C(metaclass=M):
        y, z = 11, 22
        @classmethod                         # Class meth
        def b(cls):
            return cls.x

>>> C.a()            # Call metaclass method; visible t
```

```
>>> C.x                 # Creates class data on C, accessi
33

>>> I = C()
>>> I.x, I.y, I.z
(33, 11, 22)

>>> I.b()               # Class method: sends class, not i
33
>>> I.a()               # Metaclass methods: accessible th
AttributeError: 'C' object has no attribute 'a'
```

This yields two very different ways to create class methods whose asymmetry is left as suggested pondering here.

NOTE

Plus one more: Python 3.6 added an operator-overloading method named `__init_subclass__`. This method is called whenever the containing class is *subclassed*, and it receives a single argument: the new subclass object. It provides yet another way to manage classes, though this method is nowhere near as general as class decorators or metaclasses: it's run only for a class's own subclasses and only when those subclasses are created. Similar to metaclass methods, this method is also implicitly converted to a *class method*. Because we just can't get enough of those implicit hooks!

Operator Overloading in Metaclass Methods

Just in case your head is not yet spinning, metaclasses, just like normal classes, may also employ operator overloading to make built-in operations applicable to their instance classes. The `__getitem__` indexing method in the following metaclass, for example, is a metaclass method designed to process *classes* themselves—the classes that are instances of the metaclass, not those classes' own nonclass instances:

```
>>> class M(type):
        def __getitem__(cls, i):          # Meta method
            return cls.data[i]            #  Built-ins s
                                          #  Explicit nc
>>> class C(metaclass=M):                 #  Data descri
        data = 'hack'
```

```
>>> C[0]                    # Metaclass instance names: v
'h'
>>> C.__getitem__
<bound method M.__getitem__ of <class '__main__.C'>>

>>> I = C()
>>> I.data, C.data          # Normal inheritance names: v
('hack', 'hack')
>>> I[0]
TypeError: 'C' object is not subscriptable
```

And if that's not abstruse enough, when both class and metaclass overload the *same* operator, the former applies to classes and the latter to nonclass instances:

```
>>> class C(metaclass=M):
        data = 'hack'
        def __getitem__(self, i):
            return self.data[i]

>>> I = C()
>>> I.data = 'code'
>>> C[0], I[0]              # C's [] from metaclass, I's
('h', 'c')
```

All of which leads us to the closing compare-and-contrast of the next section.

Metaclass Methods Versus Instance Methods

The inescapable conclusion of this technical novel is that metaclass methods provide a separate—and arguably redundant—way to code object behavior with classes in Python. As we've learned, the usual way to implement objects is with classes and their nonclass instances. Here's the *normal* case—with class data, constructor, regular method, and operator overloading available to instances and subclasses:

```
>>> class Employee:                                  # A '
        rate = 50                                    # Wit
        def __init__(self, name, hours):             # Plu
            self.name = name
            self.hours = hours
```

```
        def pay(self):
            print(f'
💰
 {self.name}
➡️
 ${self.rate * self.hours:,.2f}')
        def __iadd__(self, hours):
            self.hours += hours
            return self
```

As usual, we make an instance of a normal class like this by calling it with constructor arguments:

```
>>> pat = Employee('Pat', 2_000)                # A '
>>> pat.pay()                                   # Met
💰
 Pat
➡️
 $100,000.00
>>> pat += 1_000                                # Upd
>>> pat.pay()
💰
 Pat
➡️
 $150,000.00
```

And to make more instances, we simply call the class again:

```
>>> pat2 = Employee('Pat2', 1_000)              # And
>>> pat2.pay()
💰
 Pat2
➡️
 $50,000.00
>>> pat2
<__main__.Employee object at 0x10cf5c680>
```

The equivalent *metaclass* can also provide data, regular methods, and operator overloading. But per-instance constructors don't quite apply; methods receive and process a class, not instances of it; and this behavior is for subclasses and instance classes in the secondary "type" tree only:

```
>>> class MetaEmployee(type):                   # A m
        rate = 50                               # Wit
```

```
        def pay(cls):
            print(f'
💰
 {cls.name}
➡️
 ${cls.rate * cls.hours:,.2f}')
        def __iadd__(cls, hours):
            cls.hours += hours
            return cls
```

Because metaclass instances are classes, we define a new *class* per instance instead of running a call and code instance data as class attributes, though all the behavior defined in the metaclass applies to its instance classes:

```
>>> class pat(metaclass=MetaEmployee):              # A m
        name = 'Pat'                                # Wit
        hours = 2_000

>>> pat.pay()                                       # Met
💰
 Pat
➡️
 $100,000.00
>>> pat += 1_000                                    # Upd
>>> pat.pay()
💰
 Pat
➡️
 $150,000.00
```

We haven't made any normal instances here—just a *class* that nevertheless serves as an information record with both data and behavior methods. Coding differences aside, the main functional divergence in the metaclass approach is that *nonclass* instances created from such a class don't inherit any of the behavior defined in a metaclass:

```
>>> pat2 = pat()                                    # A '
>>> pat2.pay()                                      # Met
AttributeError: 'pat' object has no attribute 'pay'
```

To make additional instances in the metaclass world, we must instead code additional classes:

```
>>> class pat2(metaclass=MetaEmployee):           # Met
        name = 'Pat2'
        hours = 1_000

>>> pat2.pay()                                    # Wit
💰
 Pat2
➡️
 $50,000.00
```

Calling the metaclass with *no* arguments doesn't work at all because it makes a *class*, not an instance of a class—though calling the metaclass with full `type` arguments *does* work because it's equivalent to running a `class` statement (and yes, the fact that metaclasses are classes, too, makes some of this paragraph's logic circular, but that's just how Python works: metaclasses are special-cased for metaclass roles):

```
>>> pat2 = MetaEmployee('pat2', (), dict(name='Pat2', h
>>> pat2.pay()
💰
 Pat2
➡️
 $50,000.00
>>> pat2
<class '__main__.pat2'>
```

In sum, metaclasses allow us to implement classes that work much like the nonclass instances we've used throughout this book. As to *why* you'd want to swap one kind of instance for another with identical functionality but different coding, we'll have to defer to other resources to justify the *redundancy*. Given the many convolutions that metaclasses bring to the table, it's difficult not to see this as complexity for all in the name of rare and narrow roles. While that has been a recurring theme in both Python and this book, it's time to wrap up and move on to the conclusion.

Chapter Summary

In this chapter, we studied metaclasses, explored examples of them in action, and formalized the rules of inheritance that they extend. Along the way, we also saw how the roles of class decorators and metaclasses often intersect:

because both run at the conclusion of a `class` statement, they can sometimes be used interchangeably. We also learned how metaclass methods define behavior for classes much like the class methods we met earlier, but are limited in scope to the secondary inheritance tree searched for classes.

Since this chapter covered an advanced topic, we'll work through just a few quiz questions to review the basics (candidly, if you've made it this far in a chapter on metaclasses, you probably already deserve extra credit!). Because this is the last part of the book, we'll also forgo the end-of-part exercises. Be sure to see the appendixes that follow for Python platform pointers and the solutions to the prior parts' exercises; the last of these includes a sampling of short application-level programs for self-study.

Once you finish the quiz, you've officially reached the end of this book's technical material. The next and final chapter offers some brief thoughts about Python to wrap up the book at large and closes with some fun. We'll regroup there in the Learning Python benediction after you work through this final quiz.

Test Your Knowledge: Quiz

1. What is a metaclass?
2. How do you declare the metaclass of a class?
3. How do class decorators overlap with metaclasses for managing classes?

Test Your Knowledge: Answers

1. A metaclass is a class used to create a class. Classes are instances of the `type` class by default. Metaclasses are usually subclasses of the `type` class, which customize classes. They may redefine class-creation protocol methods like `__new__` and `__init__` to customize the class creation call issued at the end of a `class` statement. They may also define data and methods that provide behavior for their instance classes, but these are inherited only by classes, not their nonclass instances. Metaclasses can also be coded in other ways—as simple functions, for example—but they are responsible for making and returning an object for the new class. Finally, metaclasses constitute a secondary pathway for inheritance search

used for classes alone; this allows classes to ape normal instance behavior, though it bifurcates the class story for relatively rare and narrow roles.

2. Use a keyword argument in the `class` header line: `class C(metaclass=M)`. The `class` header line can also name normal superclasses before the `metaclass` keyword argument; these superclasses are searched before metaclasses for inheritance run from a class.
3. Because both are automatically triggered at the end of a `class` statement, class decorators and metaclasses can both be used to manage classes. Decorators rebind a class name to a callable's result, and metaclasses route class creation through a callable, but both hooks can be used for similar purposes. To manage classes, decorators simply augment and return the original class objects. Metaclasses augment a class after or before they create it. As noted, metaclasses can also provide behavior for their instance classes in the form of methods that are not immediately supported by decorators, though the objects returned by decorators can do anything supported by the Python language.