CHAPTER 38
Managed Attributes

This chapter expands on the *attribute interception* techniques introduced earlier, introduces another, and employs them in a handful of larger examples. Like everything in this part of the book, this chapter is classified as an advanced topic and optional reading, because most applications programmers don't need to care about the material discussed here—they can fetch and set attributes on objects without concern for attribute implementations.

Especially for tools builders, though, managing attribute access can be an important part of flexible APIs. Moreover, an understanding of the descriptor model covered here can make related tools such as slots and properties more tangible and may even be required reading if it appears in code you must use.

Why Manage Attributes?

Object attributes are central to most Python programs—they are where we often store information about the entities our scripts process. Normally, attributes are simply names for objects; a person's `name` attribute, for example, might be a simple string, fetched and set with basic attribute syntax:

```
person.name                # Fetch attribute value
person.name = value        # Change attribute value
```

In most cases, the attribute lives in the object itself or is inherited from a class from which it derives. That basic model suffices for most programs you will write in your Python career.

Sometimes, though, more flexibility is required. Suppose you've written a program to use a `name` attribute directly, but then your requirements change—for example, you decide that names must be validated or mutated with program logic when accessed. It's straightforward to code methods to manage access to the attribute's value (`valid` and `transform` are abstract and hypothetical here):

```
class Person:
    def getName(self):
        if not valid():
            raise TypeError('cannot fetch name')
        else:
            return self.name.transform()

    def setName(self, value):
         if not valid(value):
            raise TypeError('cannot change name')
        else:
            self.name = transform(value)

person = Person()
person.getName()
person.setName('value')
```

The problem with this is that it also requires changing all the places where names are used in the entire program—a possibly nontrivial task. Moreover, this approach requires the program to be aware of how values are exported: as simple names or called methods. If you begin with a method-based interface to data, clients are immune to changes; if you do not, changes can become problematic.

This issue can crop up more often than you might expect. The value of a cell in a spreadsheet-like program, for instance, might begin its life as a simple discrete value but later mutate into an arbitrary calculation. Since an object's interface should be flexible enough to support such future changes without breaking existing code, switching to methods later is less than ideal.

Inserting Code to Run on Attribute Access

A better solution would allow you to run code automatically on attribute access if needed. That's one of the main roles of managed attributes—they provide ways to add *attribute accessor* logic after the fact. More generally, they support arbitrary attribute usage modes that go beyond simple data storage.

At various points in this book, we've met Python tools that allow our scripts to dynamically compute attribute values when fetching them and validate or change attribute values when storing them. In this chapter, we're going to focus more deeply on the tools already introduced, explore other tools in this category, and study some larger use-case examples in this domain. Specifically, this chapter presents *four* accessor techniques:

1. The `property` built-in, for specifying methods to handle access to a specific attribute
2. The `__get__` and `__set__` descriptor methods, for handling access to a specific attribute and the basis for other tools such as properties and slots
3. The `__getattr__` and `__setattr__` methods, for handling undefined attribute fetches and all attribute assignments
4. The `__getattribute__` method, for handling all attribute fetches

We met these tools in Chapters 30 and 32 briefly, and in some cases, hardly at all. As you'll see here, all four techniques share goals to some degree, and it's usually possible to code a given problem using any one of them.

That said, they also differ in some important ways. For example, the first two techniques listed here apply to *specific* attributes, whereas the last two are generic enough to be used by delegation-based proxy classes that must route *arbitrary* attributes to wrapped objects. As you'll find, all four schemes also differ in both complexity and aesthetics in ways you must see in action to judge for yourself.

Besides studying the specifics behind these four attribute interception techniques, this chapter also presents an opportunity to explore programs larger than most we've seen elsewhere in this book. The `CardHolder` case study at the end, for example, should serve as a self-study example of larger classes in action. We'll also be using some of the techniques outlined here in the next chapter to code decorators, so be sure you have at least a general understanding of these topics before you move on.

Properties

Up first, the property protocol allows us to route a specific attribute's get, set, and delete operations to functions or methods we provide, enabling us to insert code to be run automatically on attribute accesses, intercept attribute deletions, and provide documentation for attributes if desired.

As introduced in Chapter 32, properties are created with the `property` built-in and are assigned to class attributes, just like method functions. Accordingly, they are inherited by subclasses and instances, like any other class attributes. Their access-interception functions are provided with the `self` instance argument, which grants access to state information and class attributes available on the subject instance.

A property manages a single, specific attribute; although it can't catch all attribute accesses generically, it allows us to control both fetch and assignment accesses and enables us to change an attribute from simple data to a computation freely without breaking existing code. As you'll see, properties are strongly related to descriptors; in fact, they are essentially a restricted form of them.

The Basics

A property is created by assigning the result of a built-in function to a class attribute:

```
attribute = property(fget, fset, fdel, doc)
```

None of this built-in's arguments are required, and all default to `None` if not passed. For the first three, this `None` means that the corresponding operation is not supported, and attempting it will raise an `AttributeError` exception automatically.

When these arguments are used, we pass `fget` a function for intercepting attribute fetches, `fset` a function for assignments, and `fdel` a function for attribute deletions. Technically, all three of these arguments accept any callable, including a class's method, having a first argument to receive the instance being qualified. When later invoked, the `fget` function returns the computed attribute value, `fset` and `fdel` return nothing (really, `None`), and all three may raise exceptions to reject access requests.

The `doc` argument receives a documentation string for the attribute if desired. If omitted, the property copies the docstring of the `fget` function, which, as usual, defaults to `None`.

This built-in `property` call returns a property object, which we assign to the name of the attribute to be managed in the class scope, where it will be inherited by every instance. As you'll learn ahead, this assignment can be automated by `@` decorator syntax, though its distributed usage may seem awkward for set and delete methods. However assigned, later accesses to the attribute automatically invoke the property's handlers.

A First Example

To demonstrate how this translates to working code, the class in Example 38-1 uses a property to trace access to an attribute named `name`; the actual stored data is named `_name` so it does not clash with the property.

Example 38-1. prop-person.py

```
class Person:
    def __init__(self, name):
        self._name = name

    def getName(self):
        print('fetch...')
        return self._name

    def setName(self, value):
        print('change...')
        self._name = value

    def delName(self):
        print('remove...')
        del self._name

    name = property(getName, setName, delName, 'name property docs')

sue = Person('Sue Jones')          # sue has a managed attribute
print(sue.name)                    # Runs getName
sue.name = 'Susan Jones'           # Runs setName
print(sue.name)
del sue.name                       # Runs delName

print('-'*20)
bob = Person('Bob Smith')          # bob inherits property too
print(bob.name)
print(Person.name.__doc__)         # Or help(Person.name)
```

This particular property doesn't do much—it simply intercepts and traces an attribute—but it serves to demonstrate the protocol. When this code is run, two instances inherit the property, just as they would any other attribute attached to their class. However, accesses to their `name` attribute are caught and managed by the code we provide:

```
$ python3 prop-person.py
fetch...
Sue Jones
change...
fetch...
Susan Jones
remove...
--------------------
fetch...
Bob Smith
name property docs
```

Like all class attributes, properties are *inherited* by both instances and lower subclasses. If we change our example as follows, for instance:

```
    class Super:
        …the original Person class code…
        name = property(getName, setName, delName, 'name property docs')

    class Person(Super):
        pass                            # Properties are inherited (class attrs)

    sue = Person('Sue Jones')
    …rest unchanged…
```

the output is the same—the `Person` subclass inherits the `name` property from `Super`, and the `sue` instance gets it from `Person`. In terms of inheritance, properties work the same as normal methods; because they have access to the `self` instance argument, they can access instance state information and methods irrespective of subclass depth, as the next section further demonstrates.

Computed Attributes

The example in the prior section simply traces attribute accesses. Usually, though, properties do much more—computing the value of an attribute dynamically when fetched, for instance, as Example 38-2 illustrates.

Example 38-2. prop-computed.py

```
class PropSquare:
    def __init__(self, start):
        self.value = start

    def getX(self):                     # On attr fetch
        return self.value ** 2

    def setX(self, value):              # On attr assign
        self.value = value

    X = property(getX, setX)            # No delete or docs

P = PropSquare(3)       # Two instances of class with property
Q = PropSquare(32)      # Each has different state information

print(P.X)              # 3 ** 2
P.X = 4
print(P.X)              # 4 ** 2
print(Q.X)              # 32 ** 2 (1024)
```

This class defines an attribute `X` that is accessed as though it were simple data, but really runs code to compute its value when fetched. The net effect triggers an implicit method call. When the code is run, the value is stored in the instance as state information, but each time we fetch it via the managed attribute, its value is automatically squared:

```
    $ python3 prop-computed.py
    9
    16
    1024
```

Notice that we've made two different instances—because property methods automatically receive a `self` argument, they have access to the state information stored in instances. In our case, this means the fetch computes the square of the subject instance's own data.

Coding Properties with Decorators

Although we're saving additional details until the next chapter, we introduced *function decorator* basics earlier, in Chapter 32. Recall that the function decorator syntax:

```
@decorator
def func(args): …
```

is automatically translated to this equivalent by Python to *rebind* the function name to the result of the `decorator` callable:

```
def func(args): …
func = decorator(func)
```

Because of this mapping, the `property` built-in can automatically serve as a decorator to define a function that will run automatically when an attribute is fetched:

```
class Person:
    @property
    def name(self): ...              # Rebinds: name = property(name)
```

When run, the decorated method is automatically passed to the first argument of the `property` built-in. This is really just alternative syntax for creating a property and rebinding the attribute name manually, but may be seen as more explicit in this role:

```
class Person:
    def name(self): …
    name = property(name)            # Manual equivalent to @property
```

Setter and deleter decorators

The preceding works naturally for property get functions, but what about other accesses? In full detail, property objects also have `getter`, `setter`, and `deleter` methods that assign the corresponding property accessor methods and return a copy of the property itself. We can use these to specify components of properties by decorating normal methods, too, though the `getter` component (along with attributes docs) is usually filled in automatically by the act of creating the property itself. Example 38-3 demos the basics.

Example 38-3. prop-person-deco.py

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):                  # name = property(name)
        'name property docs'
        print('fetch...')
        return self._name

    @name.setter
    def name(self, value):           # name = name.setter(name)
        print('change...')
        self._name = value

    @name.deleter
    def name(self):                  # name = name.deleter(name)
        print('remove...')
        del self._name
```

```
sue = Person('Sue Jones')          # sue has a managed attribute
print(sue.name)                    # Runs name getter (def name 1)
sue.name = 'Susan Jones'           # Runs name setter (def name 2)
print(sue.name)
del sue.name                       # Runs name deleter (def name 3)

print('-'*20)
bob = Person('Bob Smith')          # bob inherits property too
print(bob.name)
print(Person.name.__doc__)         # Or help(Person.name)
```

In fact, this code is equivalent to the first example in this section—decoration is just an alternative way to code properties in this case. When it's run, the results are the same:

```
$ python3 prop-person-deco.py
fetch...
Sue Jones
change...
fetch...
Susan Jones
remove...
--------------------
fetch...
Bob Smith
name property docs
```

Compared to manual assignment of `property` results, using decorators to properties in this example requires just three extra lines of code—a seemingly negligible difference. As is so often the case with alternative tools, though, the choice between the two techniques is largely subjective.

Descriptors

Very briefly previewed in Chapter 32, *descriptors* provide an alternative way to intercept attribute access; they are strongly related to the properties discussed in the prior section. Really, a property *is* a kind of descriptor—technically speaking, the `property` built-in is just a simplified way to create a specific type of descriptor that runs method functions on attribute accesses. In fact, descriptors are the underlying implementation mechanism for a variety of class tools, including both properties and slots, and play other internal roles in Python that we can safely skip here.

Functionally speaking, the descriptor protocol allows us to route a specific attribute's get, set, and delete operations to methods of a separate class's instance object that we provide. This allows us to insert code to be run automatically on attribute fetches and assignments, intercept attribute deletions, and provide documentation for the attributes if desired.

Descriptors are created as independent *classes*, and they are assigned to class attributes just like method functions. Like any other class attribute, they are inherited by subclasses and instances. Their access-interception methods are provided with both a `self` for the descriptor instance itself as well as the instance of the client class whose attribute references the descriptor object. Because of this, they can retain and use state information of their own, as well as state information of the subject instance. For example, a descriptor may call methods available in the client class, as well as descriptor-specific methods it defines.

Like a property, a descriptor manages a single, specific attribute; although it can't catch all attribute accesses generically, it provides control over both fetch and assignment accesses and allows us to change an attribute name freely from simple data to a computation without breaking existing code. If this sounds like properties, it's because it is: as you shall see, properties can be coded as descriptors directly.

Unlike properties, though, descriptors provide a more general tool. For instance, because they are coded as normal classes, descriptors have their own state, may participate in descriptor inheritance hierarchies, can use composition to aggregate objects, and provide a natural structure for coding internal methods and attribute documentation strings.

The Basics

As mentioned, descriptors are coded as separate classes and provide specially named accessor methods for the attribute access operations they wish to intercept—get, set, and deletion methods in the descriptor class are automatically run when the attribute assigned to the descriptor class instance is accessed in the corresponding way:

```
class Descriptor:
    "docstring goes here"
    def __get__(self, instance, owner): …       # Return attr value
    def __set__(self, instance, value): …       # Return nothing (None)
    def __delete__(self, instance): …           # Return nothing (None)
```

Classes with any of these methods are considered descriptors, and their methods are special when one of their instances is assigned to another class's attribute—when the attribute is accessed, these methods are automatically invoked.

If any of these methods are absent, it generally means that the corresponding type of access is not supported. Unlike properties, however, omitting a `__set__` allows the descriptor attribute's name to be assigned and thus redefined in an instance, thereby *hiding* the descriptor—to make an attribute *read-only*, you must define `__set__` to catch assignments and raise an exception.

Descriptors with `__set__` methods also have some special-case implications for inheritance that we'll largely defer until Chapter 40's coverage of metaclasses and the complete inheritance specification. In short, a descriptor with a `__set__` is known formally as a *data descriptor* and is given precedence over other names located by normal inheritance rules. The inherited descriptor for attribute `__class__`, for example, overrides the same name in an instance's namespace dictionary. This also works to ensure that data descriptors you code in your own classes take precedence over others.

Descriptor method arguments

Before we code anything realistic, let's take a brief look at some fundamentals. All three descriptor methods outlined in the prior section are passed both the descriptor class instance (`self`) and the instance of the client class to which the descriptor instance is attached (`instance`).

The `__get__` access method additionally receives an `owner` argument, specifying the class to which the descriptor instance is attached. Its `instance` argument is either the instance through which the attribute was accessed (for *`instance`*`.attr`), or `None` when the attribute is accessed through the owner class directly (for *`class`*`.attr`). The former of these generally computes a value for instance access, and the latter usually returns `self` if descriptor object access is supported.

For example, in the following REPL session, when `X.attr` is fetched, Python automatically runs the `__get__` method of the `Descriptor` class instance to which the `Subject.attr` class attribute is assigned:

```
>>> class Descriptor:
        def __get__(self, instance, owner):
            print(self, instance, owner, sep='\n')

>>> class Subject:
        attr = Descriptor()           # Descriptor instance is class attr

>>> X = Subject()
>>> X.attr
<__main__.Descriptor object at 0x104bc9b20>
<__main__.Subject object at 0x104b8a570>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x104bc9b20>
None
<class '__main__.Subject'>
```

Notice the arguments automatically passed in to the `__get__` method in the first attribute fetch—when `X.attr` is fetched, it's as though the following translation occurs (though the `Subject.attr` here doesn't invoke `__get__` again as it normally would):

```
X.attr  =>  Descriptor.__get__(Subject.attr, X, Subject)
```

The descriptor knows it is being accessed directly when its instance argument is `None`.

Read-only descriptors

As mentioned earlier, unlike properties, simply omitting the `__set__` method in a descriptor isn't enough to make an attribute read-only because the descriptor name can be assigned in an instance. In the following, the attribute assignment to `X.a` stores `a` in the instance object `X`, thereby hiding the descriptor stored in class `C`:

```
>>> class D:
        def __get__(*args): print('get')

>>> class C:
        a = D()                    # Attribute "a" is a descriptor instance

>>> X = C()
>>> X.a                            # Runs inherited descriptor __get__
get
>>> C.a
get
>>> X.a = 99                       # Stored on X, hiding C.a!
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a                            # Y still inherits descriptor
get
>>> C.a
get
```

This is the way all instance attribute assignments work in Python, and it allows classes to selectively override class-level defaults in their instances. To make a descriptor-based attribute read-only, catch the assignment in the descriptor class and raise an exception to prevent attribute assignment—when assigning an attribute that is a descriptor, Python effectively bypasses the normal instance-level assignment behavior and routes the operation to the descriptor object:

```
>>> class D:
        def __get__(*args): print('get')
        def __set__(*args): raise AttributeError('cannot set')

>>> class C:
        a = D()

>>> X = C()
>>> X.a                                 # Routed to C.a.__get__
get
>>> X.a = 99                            # Routed to C.a.__set__
AttributeError: cannot set
```

The deletion trio: Be careful not to confuse the descriptor `__delete__` method with the general `__del__` method. The former is called on attempts to delete the managed attribute name on an instance of the owner class; the latter is the general instance destructor method, run when an instance of any kind of class is about to be garbage-collected. Descriptor `__delete__` is more closely related to the `__delattr__` generic attribute deletion method we'll study later in this chapter. See Chapter 30 for more on operator-overloading methods like `__del__`.

A First Example

To see how this all comes together in more realistic code, let's get started with the same first example we wrote for properties. Example 38-4 defines a descriptor that intercepts access to an attribute named `name` in its clients. Its methods use their `instance` argument to access state information in the subject instance, where the name string is actually stored.

Example 38-4. desc-person.py

```
class Name:
    'name descriptor docs'

    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name

    def __set__(self, instance, value):
        print('change...')
        instance._name = value

    def __delete__(self, instance):
        print('remove...')
        del instance._name

class Person:
    def __init__(self, name):
        self._name = name

    name = Name()                       # Assign descriptor to attr

sue = Person('Sue Jones')               # sue has a managed attribute
```

```
print(sue.name)                     # Runs Name.__get__
sue.name = 'Susan Jones'            # Runs Name.__set__
print(sue.name)
del sue.name                        # Runs Name.__delete__

print('-'*20)
bob = Person('Bob Smith')           # bob inherits descriptor too
print(bob.name)
print(Name.__doc__)                 # Or help(Name)
```

Notice in this code how we assign an instance of our descriptor class to a *class attribute* in the client class; because of this, it is inherited by all instances of the class, just like a class's methods. Really, we *must* assign the descriptor to a class attribute like this—it won't work if assigned to a `self` instance attribute instead. When the descriptor's `__get__` method is run, it is passed three objects to define its context:

- `self` is the `Name` class instance.
- `instance` is the `Person` class instance.
- `owner` is the `Person` class.

When this code is run, the descriptor's methods intercept accesses to the attribute, much like the property version. In fact, the output is the same again:

```
$ python3 desc-person.py
fetch...
Sue Jones
change...
fetch...
Susan Jones
remove...
--------------------
fetch...
Bob Smith
name descriptor docs
```

Also like in the property example, our descriptor class instance is a class attribute and thus is *inherited* by all instances of the client class and any subclasses. If we change the `Person` class in our example to the following, for instance, the output of our script is the same:

```
…
class Super:
    def __init__(self, name):
        self._name = name

    name = Name()

class Person(Super):                # Descriptors are inherited (class attrs)
   pass
…
```

Also, note that when a descriptor class is not useful outside the client class, it's perfectly reasonable to embed the descriptor's definition inside its client syntactically. Here's what our example looks like if we use a *nested class*:

```
class Person:
    def __init__(self, name):
        self._name = name

    class Name:                                   # Using a nested class
        'name descriptor docs'

        def __get__(self, instance, owner):
            …same…

        def __set__(self, instance, value):
            …same…

        def __delete__(self, instance):
            …same…

    name = Name()
```

When coded this way, `Name` becomes a local variable in the scope of the `Person` class statement, such that it won't clash with any names outside the class. This version works the same as the original—we've simply moved the descriptor class definition into the client class's scope—but the last line of the testing code must change to fetch the docstring from its new location (per unlisted file *desc-person-nested.py* in the example's package):

```
…
print(Person.Name.__doc__)     # Differs: not Name.__doc__ outside class
```

Computed Attributes

As was the case when using properties, our first descriptor example of the prior section didn't do much—it simply printed trace messages for attribute accesses as a demo. In practice, descriptors can also be used to compute attribute values each time they are fetched. Example 38-5 illustrates—it's a rehash of the same example we coded for properties but uses a descriptor to automatically square an attribute's value each time it is fetched.

Example 38-5. desc-computed.py

```
class DescSquare:
    def __init__(self, start):                  # Each desc has own state
        self.value = start

    def __get__(self, instance, owner):         # On attr fetch
        return self.value ** 2

    def __set__(self, instance, value):         # On attr assign
        self.value = value                      # No delete or docs

class Client1:
    X = DescSquare(3)          # Assign descriptor instance to class attr

class Client2:
    X = DescSquare(32)         # Another instance in another client class
                               # Could also code two instances in same class
c1 = Client1()
c2 = Client2()

print(c1.X)                    # 3 ** 2
c1.X = 4
```

```
print(c1.X)                     # 4 ** 2
print(c2.X)                     # 32 ** 2 (1024)
```

When run, the output of this example is the same as that of the original property-based version, but here a descriptor class object is intercepting the attribute accesses instead of a property:

```
$ python3 desc-computed.py
9
16
1024
```

Using State Information in Descriptors

If you closely study the two descriptor examples we've written so far, you might notice that they get their information from different places—the first (the `name` attribute example) uses data stored on the client *instance*, and the second (the attribute squaring example) uses data attached to the *descriptor* object itself (a.k.a. `self`). In fact, descriptors can use *both* instance state and descriptor state, or any combination thereof:

- *Descriptor state* is used to manage either data internal to the workings of the descriptor or data that spans all instances. It can vary per attribute appearance (often per client class).
- *Instance state* records information related to and possibly created by the client class. It can vary per client-class instance (that is, per application object).

In other words, descriptor state is per-descriptor data, and instance state is per-client-instance data. As usual in OOP, you must choose state carefully. For example, you would not normally use *descriptor* state to record employee names since each client instance requires its own value—if stored in the descriptor, each client class instance will effectively share the same single copy. On the other hand, you would not usually use *instance* state to record data pertaining to descriptor implementation internals—if stored in each instance, there would be multiple varying copies.

Descriptor methods may use either state form, but descriptor state sometimes makes it unnecessary to use special naming conventions to avoid name collisions in the instance for data that is not instance specific. For example, the descriptor in Example 38-6 attaches information to its own instance, so it doesn't clash with that on the client class's instance—but also shares that information between two client instances.

Example 38-6. desc-state-desc.py

```
class DescState:                          # Use descriptor state
    def __init__(self, value):
        self.value = value

    def __get__(self, instance, owner):   # On attr fetch
        print('DescState get')
        return self.value * 10

    def __set__(self, instance, value):   # On attr assign
        print('DescState set')
        self.value = value

# Client class
class CalcAttrs:
    X = DescState(2)                      # Descriptor class attr
    Y = 3                                 # Class attr
    def __init__(self):
        self.Z = 4                         # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                 # X is computed, others are not
obj.X = 5                                  # X assignment is intercepted
CalcAttrs.Y = 6                            # Y reassigned in class
obj.Z = 7                                  # Z assigned in instance
print(obj.X, obj.Y, obj.Z)

obj2 = CalcAttrs()                         # But X uses shared data, like Y!
print(obj2.X, obj2.Y, obj2.Z)
```

This code's internal `value` information lives only in the *descriptor*, so there won't be a collision if the same name is used in the client's instance. Notice that only the descriptor attribute is managed here—get and set accesses to `X` are intercepted, but accesses to `Y` and `Z` are not (`Y` is attached to the client class and `Z` to the instance). When this code is run, `X` is computed when fetched, but its value is also the same for all client instances because it uses descriptor-level state:

```
$ python3 desc-state-desc.py
DescState get
20 3 4
DescState set
DescState get
50 6 7
DescState get
50 6 4
```

It's also feasible for a descriptor to store or use an attribute attached to the client class's *instance* instead of itself. Crucially, unlike data stored in the descriptor itself, this allows for data that can vary per client class instance. The descriptor in Example 38-7 assumes the instance has an attribute `_X` attached by the client class and uses it to compute the value of the attribute it represents.

Example 38-7. desc-state-inst.py

```
class InstState:                           # Using instance state
    def __get__(self, instance, owner):
        print('InstState get')             # Assume set by client class
        return instance._X * 10

    def __set__(self, instance, value):
        print('InstState set')
        instance._X = value

# Client class
class CalcAttrs:
    X = InstState()                        # Descriptor class attr
    Y = 3                                  # Class attr
    def __init__(self):
        self._X = 2                        # Instance attr
        self.Z  = 4                        # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                 # X is computed, others are not
obj.X = 5                                  # X assignment is intercepted
CalcAttrs.Y = 6                            # Y reassigned in class
obj.Z = 7                                  # Z assigned in instance
print(obj.X, obj.Y, obj.Z)
```

```
obj2 = CalcAttrs()                          # But X differs now, like Z!
print(obj2.X, obj2.Y, obj2.Z)
```

Here, `X` is assigned to a descriptor as before that manages accesses. The new descriptor here, though, has no information itself, but it uses an attribute assumed to exist in the instance—that attribute is named `_X`, to avoid collisions with the name of the descriptor itself. When this version is run, the results are similar, but the value of the descriptor attribute can vary per client instance due to the differing state policy:

```
$ python3 desc-state-inst.py
InstState get
20 3 4
InstState set
InstState get
50 6 7
InstState get
20 6 4
```

Both descriptor and instance state have roles. In fact, this is a general advantage that descriptors have over properties—because they have state of their own, they can easily retain data internally without adding it to the namespace of the client instance object. As a summary, the following uses *both* state sources—its `self.data` retains per-attribute information, while its `instance.data` can vary per client instance:

```
>>> class DescBoth:
        def __init__(self, data):
            self.data = data
        def __get__(self, instance, owner):
            return f'{self.data}, {instance.data}'
        def __set__(self, instance, value):
            instance.data = value

>>> class Client:
        def __init__(self, data):
            self.data = data
        managed = DescBoth('hack')

>>> I = Client('code')
>>> I.managed                       # Show both data sources
'hack, code'
>>> I.managed = 'HACK'              # Change instance data
>>> I.managed
'hack, HACK'
```

We'll revisit the implications of this choice in a case study later in this chapter. Before we move on, recall from Chapter 32's coverage of *slots* that we can access "virtual" attributes like properties and descriptors with tools like `dir` and `getattr`, even though they don't exist in the instance's namespace dictionary. Whether you *should* access these this way probably varies per program—properties and descriptors may run arbitrary computation and may be less obviously instance "data" than slots:

```
>>> I.__dict__
{'data': 'HACK'}
>>> [x for x in dir(I) if not x.startswith('__')]
['data', 'managed']

>>> getattr(I, 'data')
'HACK'
>>> getattr(I, 'managed')
```

```
'hack, HACK'

>>> for attr in (x for x in dir(I) if not x.startswith('__')):
        print(f'{attr} => {getattr(I, attr)}')

data => HACK
managed => hack, HACK
```

The more generic `__getattr__` and `__getattribute__` tools we'll explore soon are not designed to support this functionality: because they have no class-level attributes, their "virtual" attribute names do not appear in `dir` results (per Chapter 31, a `__dir__` can provide a `dir` result, but it's optional and uncommon). In exchange, they are also not limited to specific attribute names coded as properties or descriptors—tools that share even more than this behavior, as the next section explains.

How Properties and Descriptors Relate

As mentioned earlier, properties and descriptors are strongly related—the `property` built-in is just a convenient way to create a descriptor. Now that you know how both work, you should also be able to see that it's possible to simulate the `property` built-in with a descriptor class, as demoed by Example 38-8.

Example 38-8. prop-desc-equiv.py

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel                                 # Save unbound methods
        self.__doc__ = doc                               # or other callables

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get attribute")
        return self.fget(instance)                       # Pass instance to self
                                                         # in property accessors
    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(instance)

class Person:
    def getName(self):
        print('getName...')
    def setName(self, value):
        print('setName...')
    name = Property(getName, setName)                    # Use like property()

x = Person()
x.name
```

```
x.name = 'Pat'
del x.name
```

This Property class catches attribute accesses with the descriptor protocol and routes requests to functions or methods passed in and saved in descriptor state when the class's instance is created. Attribute fetches, for example, are routed from the Person class, to the Property class's __get__ method, and back to the Person class's getName. With descriptors, this "just works":

```
    $ python3 prop-desc-equiv.py
    getName...
    setName...
    AttributeError: can't delete attribute
```

Note that this descriptor class equivalent only handles basic property usage, though; to use @ *decorator syntax* to also specify set and delete operations, we'd have to extend our Property class with setter and deleter methods, which would save the decorated accessor function and return the property object (self should suffice). Since the property built-in already does this, we'll omit a formal coding of this extension here.

Descriptors and slots and more

You can also probably now, at least in part, imagine how descriptors are used to implement Python's *slots* extension: instance attribute dictionaries are avoided by creating class-level descriptors that intercept slot name access and map those names to sequential storage space in the instance. Unlike the explicit property call, though, much of the magic behind slots is orchestrated at class creation time both automatically and implicitly when a __slots__ attribute is present in a class.

See Chapter 32 for more on slots—and why they're not recommended except in pathological use cases. Descriptors are also used for other class tools, but we'll omit further internals details here; see Python's manuals and its open source code for more details.

Descriptor cliff-hangers: In Chapter 39, we'll also make use of descriptors to implement function *decorators* that apply to both functions and methods. As you'll see there, because descriptors receive both descriptor and subject class instances they work well in this role, though nested functions are often a conceptually simpler solution. In addition, Chapter 39 deploys descriptors as one way to intercept *built-in operation* method fetches, and Chapter 40 formalizes data descriptors' precedence in the full *inheritance* model noted earlier: with a __set__, descriptors override other names and are thus fairly binding—they cannot be hidden by names in instance dictionaries.

__getattr__ and __getattribute__

So far, we've studied properties and descriptors—tools for managing specific attributes. The __getattr__ and __getattribute__ operator-overloading methods provide still other ways to intercept attribute fetches for class instances. Like properties and descriptors, they allow us to insert code to be run automatically when attributes are accessed. As shown here, though, these two methods can also be used in more general ways. Because they intercept arbitrary names, they can apply in broader roles, but may also incur extra calls in some contexts, and are too dynamic to register in dir results without help.

This form of attribute-fetch interception comes in two flavors, coded with two different methods:

- `__getattr__` is run for *undefined* attributes—because it is run only for attributes not stored on an instance or inherited from one of its classes, its use is straightforward.
- `__getattribute__` is run for *every* attribute—because it is all-inclusive, you must be cautious when using this method to avoid recursive loops by passing attribute accesses to a superclass.

We met the first of these in Chapter 30. These two methods are representatives of a set of attribute interception methods that also includes `__setattr__` and `__delattr__`. Because these methods have similar roles, though, we will generally treat them all as a single topic here.

Unlike properties and descriptors, these methods are usually considered part of Python's *operator-overloading* protocol—specially named methods of a class, inherited by subclasses, and run automatically when instances are used in the associated built-in operation (here, attribute fetch). Like all normal methods of a class, they each receive a first `self` argument when called, giving access to both instance state information and other methods of their hosting class.

The `__getattr__` and `__getattribute__` methods are also more *generic* than properties and descriptors—they can be used to intercept access to any (or even all) instance attribute fetches, not just a single specific name. Because of this, these two methods are well suited to general *delegation* coding patterns—they can implement wrapper (a.k.a. *proxy*) objects that manage all attribute accesses for an embedded object. By contrast, we must define one property or descriptor for every attribute we wish to intercept. As covered ahead, this delegation role is limited somewhat for built-in operations but still applies to all named methods in a wrapped object's interface.

Finally, these two methods are more *narrowly focused* than the alternatives we considered earlier: they intercept attribute fetches only, not assignments. To also catch attribute changes by assignment, we must code a `__setattr__` method—an operator-overloading method run for every attribute assignment, which must take care to avoid recursive loops by routing attribute assignments through the instance namespace dictionary or a superclass method. Although less common, we can also code a `__delattr__` overloading method (which must avoid looping in the same way) to intercept attribute deletions. By contrast, properties and descriptors catch get, set, and delete operations by design.

`__getattr__` and `__setattr__` were introduced in Chapters 30 and 32, and `__getattribute__` was mentioned briefly in Chapter 32. Here, we'll expand on their usage and study their roles in larger contexts.

The Basics

In short, if a class defines or inherits the following methods, they will be run automatically when an instance is used in the operation described by the comments to the right:

```
def __getattr__(self, name):        # On undefined attribute fetch [obj.name]
def __getattribute__(self, name):   # On all attribute fetch [obj.name]
def __setattr__(self, name, value): # On all attribute assignment [obj.name=value]
def __delattr__(self, name):        # On all attribute deletion [del obj.name]
```

In these, `self` is the subject instance object as usual, `name` is the string name of the attribute being accessed, and `value` is the object being assigned to the attribute. The two get methods normally return an attribute's value, and the other two return nothing (`None`). All can raise exceptions to signal prohibited access.

For example, to catch every attribute fetch, we can use either of the first two previous methods, and to catch every attribute assignment we can use the third. The following uses `__getattr__` for fetches:

```
class Catcher:
    def __getattr__(self, name):
        print('Get:', name)
    def __setattr__(self, name, value):
        print('Set:', name, value)

X = Catcher()
X.job                                # Prints "Get: job"
X.pay                                # Prints "Get: pay"
X.pay = 'bread'                      # Prints "Set: pay bread"
```

Using `__getattribute__` works exactly the same in this specific case but has subtle looping potential which we'll take up in the next section:

```
class Catcher:                                # On all attribute fetches
    def __getattribute__(self, name):         # Works same as getattr here
        print('Get:', name)                   # But prone to loops in general
    ...rest unchanged...
```

Such a coding structure can be used to implement the *delegation* design pattern we met earlier in Chapter 31. Because all attributes are routed to interception methods generically, we can validate and pass them along to embedded, managed objects. As a refresher, the following class, borrowed from Chapter 31, traces *every* attribute fetch made to another object passed to the wrapper (proxy) class:

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

X = Wrapper([1, 2, 3])
X.append(4)                          # Prints "Trace: append"
print(X.wrapped)                     # Prints "[1, 2, 3, 4]"
```

There is no such analog for properties and descriptors, short of coding accessors for *every* attribute present in *every* wrapped object. On the other hand, when such generality is not required, generic accessor methods may incur additional calls for assignments in some contexts—a trade-off described in Chapter 30 and mentioned in the context of the case study example we'll explore at the end of this chapter.

Avoiding loops in attribute interception methods

These methods are generally straightforward to use. Their most complex aspect is the potential for *looping* (a.k.a. recursing). Because `__getattr__` is called for undefined attributes only, it can freely fetch other attributes within its own code. However, because `__getattribute__` and `__setattr__` are run for *all* attributes, their code must be careful when accessing other attributes to avoid calling themselves again and triggering a recursive loop.

For example, another attribute fetch run inside a `__getattribute__` method's code like the following will trigger `__getattribute__` again—and the code will usually loop until memory is exhausted:

```
    def __getattribute__(self, name):
        x = self.other                                # LOOPS!
```

Technically, this method is even more loop-prone than this may imply—a `self` attribute reference run *anywhere* in a class that defines this method will trigger `__getattribute__` and also has the potential to loop, depending on the class's logic. This is normally desired behavior—intercepting every attribute fetch is this method's purpose, after all—but you should be aware that this method catches *all* attribute fetches wherever they are coded. When coded within `__getattribute__` itself, this almost always causes a loop.

To avoid this loop, route the fetch through a higher superclass instead to skip this level's version—because the `object` class is always a superclass to every class, it serves well in this role:

```
def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other')     # Force higher to avoid me
```

For `__setattr__`, the situation is similar, as summarized in Chapter 30—assigning *any* attribute inside this method triggers `__setattr__` again and may create a similar loop:

```
def __setattr__(self, name, value):
    self.other = value                             # Recurs (and might LOOP!)
```

Here too, `self` attribute assignments *anywhere* in a class defining this method trigger `__setattr__` as well, though the potential for looping is much stronger when they show up in `__setattr__` itself. To work around this problem, you can assign the attribute as a key in the instance's `__dict__` namespace dictionary instead. This avoids direct attribute assignment:

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value                 # Use attr dict to avoid me
```

Alternatively, `__setattr__` can also pass its own attribute assignments to a higher superclass to avoid looping, just like `__getattribute__`. In fact, this scheme is sometimes preferred when wrapped classes use *slots*, *properties*, or other "virtual" attributes that live on classes instead of instances—and in the case of slots, may preclude `__dict__`:

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value)       # Force higher to avoid me
```

This book's `__setattr__` examples often use `__dict__` for smaller demos anyhow, just because their parameters are known. By contrast, though, we *cannot* use the `__dict__` trick to avoid loops in `__getattribute__`:

```
def __getattribute__(self, name):
    x = self.__dict__['other']                     # Loops!
```

If this is coded, fetching the `__dict__` attribute itself triggers `__getattribute__` again—causing a recursive loop and an immediate fail. Strange but true!

The `__delattr__` method is less commonly used in practice, but when it is, it is called for *every* attribute deletion, just as `__setattr__` is called for every attribute assignment. When using this method, you must avoid loops when deleting attributes by the same techniques: namespace dictionaries operations or superclass method calls.

A First Example

Generic attribute management is not nearly as complicated as the prior section may have implied. To see how to put these ideas to work, Example 38-9 is the same first example we used for properties and descriptors in action again, this time implemented with attribute operator-overloading methods. Because these methods are so generic, we test attribute names here to know when a managed attribute is being accessed; others are allowed to pass normally.

Example 38-9. getattr-person.py

```
class Person:
    def __init__(self, name):                # On [Person()]
        self._name = name                    # Triggers __setattr__!

    def __getattr__(self, attr):             # On [obj.undefined]
        print('get: ' + attr)
        if attr == 'name':                   # Intercept name: not stored
            return self._name                # Does not loop: real attr
        else:                                # Others are errors
            raise AttributeError(attr)

    def __setattr__(self, attr, value):      # On [obj.any = value]
        print('set: ' + attr)
        if attr == 'name':
            attr = '_name'                   # Set internal name
        self.__dict__[attr] = value          # Avoid looping here

    def __delattr__(self, attr):             # On [del obj.any]
        print('del: ' + attr)
        if attr == 'name':
            attr = '_name'                   # Avoid looping here too
        del self.__dict__[attr]              # but much less common

sue = Person('Sue Jones')            # sue has a managed attribute
print(sue.name)                      # Runs __getattr__
sue.name = 'Susan Jones'             # Runs __setattr__
print(sue.name)
del sue.name                         # Runs __delattr__

print('-'*20)
bob = Person('Bob Smith')            # bob's attrs work like sue's
print(bob.name)
#print(Person.name.__doc__)          # No direct equivalent here!
```

When this code is run, the same sort of output is produced, but this time it reflects our generic attribute-interception methods responding to Python's normal operator-overloading mechanism:

```
$ python3 getattr-person.py
set: _name
get: name
Sue Jones
set: name
get: name
Susan Jones
del: name
--------------------
set: _name
get: name
Bob Smith
```

Notice how the attribute assignment in the `__init__` constructor triggers `__setattr__` too—this method catches *every* instance-attribute assignment, even those anywhere within the class itself, and those to underlying attributes like `_name`. Also note that, unlike with properties and descriptors, there's no direct notion of specifying *documentation* for our attribute here; managed attributes exist within the code of our interception methods, not as distinct objects.

Using __getattribute__

To achieve exactly the same results with `__getattribute__`, replace `__getattr__` in Example 38-9 with the differing code in Example 38-10. Because it catches *all* attribute fetches, this version must be careful to avoid looping by passing new fetches to a superclass, and it can't generally assume unknown names are errors.

Example 38-10. getattribute-person.py (differing part)

```
# Replace just __getattr__ with this

def __getattribute__(self, attr):                # On [obj.any]
    print('get: ' + attr)
    if attr == 'name':                           # Intercept all names
        attr = '_name'                           # Map to internal name
    return object.__getattribute__(self, attr)   # Avoid looping here
```

When run with this change, the output is similar, but we get an extra `__getattribute__` call for the fetch of `__dict__` in `__setattr__` (the first time originating in `__init__`):

```
$ python3 getattribute-person.py
set: _name
get: __dict__
get: name
Sue Jones
set: name
get: __dict__
get: name
Susan Jones
del: name
get: __dict__
--------------------
set: _name
get: __dict__
get: name
Bob Smith
```

This example is equivalent to that coded for properties and descriptors, but it's a bit artificial, and it doesn't really highlight these tools' assets. Because they are generic, `__getattr__` and `__getattribute__` are probably more commonly used in delegation-base code (as sketched earlier), where attribute access is validated and routed to an embedded object. Where just a *single* attribute must be managed, properties and descriptors might do as well or better, and avoid extra calls for unmanaged attributes.

Computed Attributes

As before, our prior example doesn't really do anything but trace attribute fetches; it's not much more work to compute an attribute's value when fetched. As for properties and descriptors, Example 38-11 creates a virtual attribute `X` that runs a calculation when fetched.

Example 38-11. getattr-computed.py

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                          # Triggers __setattr__!

    def __getattr__(self, attr):                    # On undefined attr fetch
        if attr == 'X':
            return self.value ** 2                  # value is not undefined
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):             # On all attr assignments
        if attr == 'X':
            attr = 'value'
        self.__dict__[attr] = value

A = AttrSquare(3)        # 2 instances of class with overloading
B = AttrSquare(32)       # Each has different state information

print(A.X)               # 3 ** 2
A.X = 4
print(A.X)               # 4 ** 2
print(B.X)               # 32 ** 2 (1024)
```

Running this code results in the same output that we got earlier when using properties and descriptors, but this script's mechanics are based on generic attribute interception methods:

```
$ python3 getattr-computed.py
9
16
1024
```

Using __getattribute__

As before, we can achieve the same effect with `__getattribute__` instead of `__getattr__`. Example 38-12 replaces the fetch method with a `__getattribute__` and changes the `__setattr__` assignment method to avoid looping by using direct `object` superclass method calls instead of `__dict__` keys.

Example 38-12. getattribute-computed.py

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                # Triggers __setattr__!

    def __getattribute__(self, attr):     # On all attr fetches
        if attr == 'X':
            return self.value ** 2        # Triggers __getattribute__ again!
        else:
            return object.__getattribute__(self, attr)

    def __setattr__(self, attr, value):   # On all attr assignments
        if attr == 'X':
            attr = 'value'
        object.__setattr__(self, attr, value)

…self-test code same as Example 38-11…
```

When this version is run, the results are the same again so we won't relist them here. Notice, though, the implicit and subtle routing going on inside this class's methods:

- `self.value=start` inside the constructor triggers `__setattr__`.
- `self.value` inside `__getattribute__` triggers `__getattribute__` again.

In fact, `__getattribute__` is run *twice* each time we fetch attribute `X`. This doesn't happen in the `__getattr__` version because the `value` attribute is not undefined (and hence skips the method). If you care about speed and want to avoid this, change `__getattribute__` to use the superclass to fetch `value` as well:

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

Of course, this still incurs a call to the superclass method but not an additional recursive call before we get there. If that's confusing, add `print` calls to these methods to trace how and when they run.

__getattr__ and __getattribute__ Compared

To summarize the coding differences between `__getattr__` and `__getattribute__`, Example 38-13 uses both to implement three attributes—`attr1` is a class attribute, `attr2` is an instance attribute, and `attr3` is a virtual managed attribute computed when fetched.

Example 38-13. getattr-v-getattribute.py

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):            # On undefined attrs only
        print('get:', attr)                 # Not on attr1: inherited from class
        if attr == 'attr3':                 # Not on attr2: stored on instance
            return 3
        else:
            raise AttributeError(attr)

X = GetAttr()
print(X.attr1)
print(X.attr2)
print(X.attr3)
print('-'*20)

class GetAttribute:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):       # On all attr fetches
        print('get:',  attr)                # Use superclass to avoid looping here
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)

X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)
```

When run, the `__getattr__` version intercepts only `attr3` accesses because it is undefined. The `__getattribute__` version, on the other hand, intercepts all attribute fetches and must route those it does not manage to the superclass fetcher to avoid loops:

```
$ python3 getattr-v-getattribute.py
1
2
get: attr3
3
--------------------
get: attr1
1
get: attr2
2
get: attr3
3
```

Although `__getattribute__` can catch more attribute fetches than `__getattr__`, in practice they are often just variations on a theme—if attributes are not physically stored, the two have the same effect.

Management Techniques Compared

To summarize the coding differences in all four attribute-management schemes we've just explored, let's quickly step through a somewhat more comprehensive computed-attribute example using each technique. The first version, Example 38-14, uses *properties* to intercept and calculate attributes named `square` and `cube`. Notice how their base values are stored in names that begin with an underscore so they don't clash with the names of the properties themselves.

Example 38-14. all_four_props.py

```
"Two dynamically computed attributes with properties"

class Powers:
    def __init__(self, square, cube):
        self._square = square                    # _square is the base value
        self._cube   = cube                      # square is the property name

    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)      # Or @property decorator

    def getCube(self):
        return self._cube ** 3
    cube = property(getCube)                     # Likewise
```

To do the same with *descriptors*, Example 38-15 defines the attributes with complete classes. Note that these descriptors store base values as instance state, so they must use leading underscores again so as not to clash with the names of descriptors; as called out by the final example of this chapter, we could avoid this renaming requirement by storing base values as descriptor state instead, but that doesn't as directly address data that must vary per client-class instance.

Example 38-15. all_four_desc.py

```
"Same, but with descriptors (per-instance state)"

class DescSquare:
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value

class DescCube:
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers:
    square = DescSquare()
    cube   = DescCube()
    def __init__(self, square, cube):
        self._square = square                # "self.square = square" works too,
        self._cube   = cube                  # because it triggers desc __set__!
```

To achieve the same result with `__getattr__` fetch interception, Example 38-16 again stores base values with underscore-prefixed names so that accesses to managed names are undefined and thus invoke its method; it also needs to code a `__setattr__` to intercept assignments and take care to avoid its potential for looping.

Example 38-16. all_four_getattr.py

```
"Same, but with generic __getattr__ undefined-attribute interception"

class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube   = cube

    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('unknown attr:' + name)

    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value           # Or use object
        else:
            self.__dict__[name] = value
```

The final option in Example 38-17, coding with `__getattribute__`, is similar to the prior version. Because it catches every attribute now, though, it must also route base value fetches to a superclass to avoid looping or extra calls—fetching `self._square` directly works too, but runs a second `__getattribute__` call.

Example 38-17. all_four_getattribute.py

```
"Same, but with generic __getattribute__ all-attribute interception"

class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube   = cube

    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        elif name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        else:
            return object.__getattribute__(self, name)

    def __setattr__(self, name, value):
        if name == 'square':
            object.__setattr__(self, '_square', value)   # Or use __dict__
        else:
            object.__setattr__(self, name , value)
```

To test, the following REPL session loops through a list of all four modules' name strings and imports and fetches classes along the way. Each technique takes a different form in code, but all four produce the same result when run:

```
>>> from importlib import import_module
>>> mods = [f'all_four_{M}' for M in ('props', 'desc', 'getattr', 'getattribute')]
>>> for modname in mods:
        module = import_module(modname)    # Import by name string
        X = module.Powers(3, 4)            # This module's class (print to see)
        print(X.square)                    # 3 ** 2 = 9
        print(X.cube)                      # 4 ** 3 = 64
        X.square = 5
        print(X.square)                    # 5 ** 2 = 25

9
64
25
…repeated four times…
```

For more on how these alternatives compare, and other coding options, stay tuned for a more realistic application of them in the attribute-validation example ahead. First, though, we need to take a short side trip to study a pitfall associated with two of these tools—the generic attribute interceptors.

Intercepting Built-in Operation Attributes

If you've been reading this book linearly, some of this section is elaboration on earlier notes, especially the sidebar "Delegating Built-ins—or Not" on page 698. When `__getattr__` and `__getattribute__` were introduced here, it was stated that they intercept undefined- and all-attribute fetches, respectively, which makes them ideal for delegation-based coding patterns.

While this is true for both *normally named* and *explicitly fetched* attributes, their behavior needs some additional clarification. Specifically, the implicit method-name fetches of *built-in operations* will never automatically be routed to either of these two attribute-interceptor methods. This means that operator-overloading method calls cannot be delegated to wrapped objects unless wrapper classes somehow redefine these methods themselves.

For example, attribute fetches for the `__str__`, `__add__`, and `__getitem__` methods run *implicitly* by printing, `+` expressions, and indexing, respectively, are not routed to either `__getattr__` or `__getattribute__`. Instead, such methods are looked up in classes, and skip the instance and its attribute interceptors. Hence, there is no direct way to generically catch and delegate built-in operations like these.

This was a Python 3.X bifurcation, whose purported rationale involved metaclasses and optimization of built-in operations. Whatever its basis, all attributes—both `__X__` and other—are still dispatched through the instance's interceptor methods when accessed *explicitly* by name, so this qualifies as a glaring inconsistency: *`X`*`.__add__` runs `__getattr__`, but *`X`*`+`*`Y`*, which uses *`X`*`.__add__`, does not. The net effect complicates delegation-based code.

The good news is that wrapper classes can work around this constraint by redefining operator-overloading methods in the wrapper itself, in order to catch and delegate calls. These extra methods can be added either manually, with tools, or by definition in, and inheritance from, common superclasses. It's more work for delegation classes when operator-overloading methods are part of a wrapped object's interface, but it's not a showstopper.

As a demo of the issue, consider the code in Example 38-18, which tests various attribute types and built-in operations on instances of classes containing `__getattr__` and `__getattribute__` methods.

Example 38-18. getattr-builtins.py

```
class GetAttr:
    cattr = 88                      # Attrs stored on class and instance
    def __init__(self):             # These skip getattr, but not getattribute
       self.iattr = 77

    def __len__(self):              # Redefine for len(): doesn't run getattr
        print('__len__: 66')
        return 66

    def __getattr__(self, attr):    # Provide __str__ if asked, else dummy func
        print('getattr:', attr)     # Never run for __str__: inherited from object
        if attr == '__str__':
            return lambda *args: '[Getattr str]'
        else:
            return lambda *args: None

class GetAttribute:
    cattr = 88                      # Similar, but catch all attributes
    def __init__(self):             # Except implicit fetches for built-in ops
        self.iattr = 77

    def __len__(self):              # Redefine for len(): doesn't run getattribute
        print('__len__: 66')        # But explicit fetches of inherited __str__ do
        return 66

    def __getattribute__(self, attr):
        print('getattribute:', attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'
        else:
            return lambda *args: None

for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))
```

```
X = Class()

# Defined attributes trigger getattribute but not getattr

X.cattr                  # Class attr    (defined - skips getattr)
X.iattr                  # Instance attr (defined - skips getattr)
X.other                  # Missing attr
len(X)                   # __len__ defined explicitly: moot

# Built-in ops do not invoke either getattr or getattribute
# No defaults are inherited for these from object superclass

try:    X[0]             # Tries to invoke __getitem__
except: print('fail []')
try:    X + 99           # Ditto, __add__
except: print('fail +')
try:    X()              # Ditto, __call__
except: print('fail ()')

# But explicit calls invoke both catchers

X.__getitem__(0)
X.__add__(99)
X.__call__()

# The implied object superclass defines a __str__ that precludes getattr
# But the absolute getattribute is not called for implicit fetches either

print(X.__str__())       # __str__: explicit call => only __getattr__ skipped
print(X)                 # __str__: implicit via built-in => both skipped
```

This file runs the same set of tests on each of its classes in turn. Match its following output with its tests and comments to see how it works. In short, neither `__getattr__` nor `__getattribute__` are run for any of the operator-overloading names invoked by built-in operations because such names are looked up in classes only:

```
$ python3 getattr-builtins.py

GetAttr==========================================
getattr: other
__len__: 66
fail []
fail +
fail ()
getattr: __getitem__
getattr: __add__
getattr: __call__
<__main__.GetAttr object at 0x10f76f020>
<__main__.GetAttr object at 0x10f76f020>

GetAttribute=====================================
getattribute: cattr
getattribute: iattr
getattribute: other
__len__: 66
fail []
fail +
fail ()
getattribute: __getitem__
getattribute: __add__
```

```
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x10f74c440>
```

More generally, all *explicit* method-name attribute fetches are always routed to both attribute-interception methods, but none of the *implicit* operator-overloading methods trigger either attribute-interception method when their attributes are fetched by built-in operations. Salient points in this demo worth calling out:

- `__str__` access fails to be caught twice by `__getattr__`: once for the built-in `print`, and once for explicit fetches because a default is inherited from the built-in `object` implied above every topmost class.
- `__str__` fails to be caught only once by the `__getattribute__` catchall—during the built-in `print` operation. Explicit fetches bypass the inherited `__str__` and run `__getattribute__`.
- `__call__` fails to be caught in both schemes for built-in call expressions, but it is intercepted by both when fetched explicitly; unlike `__str__`, there is no inherited `__call__` default in `object` to defeat `__getattr__` in explicit fetches. The same goes for the `__add__` of `+` operations.
- `__len__` is handled by both classes because it is an explicitly defined method in the classes themselves—though its name is not routed to either `__getattr__` or `__getattribute__` if we delete the classes' `__len__` methods because the `len` built-in skips them as usual.

Again, the net effect is that operator-overloading methods implicitly run by built-in operations are never routed through either attribute interception method. Python begins the search for such attributes in *classes* and skips instance lookup mechanisms entirely. Normally, named attributes and explicit fetches start with the instance instead.

For a more realistic example of this phenomenon's impact on delegation classes, stay tuned for Chapter 39's `Private` decorator—along with its coverage of multiple reusable *workarounds*.

Revisiting Chapter 28's delegation example

As a coda, you should also now be able to work out why the `Manager` class coded in Example 28-11 of Chapter 28 had to code a `__repr__` to route printing requests to its wrapped object. Just like `__str__` in our demo, `object` provides a default `__repr__`, which would prevent `print` operations from invoking a `__getattr__`. Technically speaking, `object` defines both `__str__` and `__repr__`, but its `__str__` simply calls `__repr__`.

That said, `object`'s defaults are largely a moot point: like all built-in operations, `print` bypasses both `__getattr__` and `__getattribute__`, as it did for `__getattribute__` in our demo. Hence, a `__repr__` is required by *both* the `object` default and the built-in's behavior.

Again, fixes for delegating built-ins are in Chapter 39 (unless we run out of underscores before that!).

Example: Attribute Validations

To close out this chapter, let's turn to a more realistic example, coded in all four of our attribute management schemes. The example we will use defines a `CardHolder` object with four attributes, three of which are managed. The managed attributes validate or transform values when fetched or stored. All four versions produce the same results for the same test code, but they implement their attributes in very different ways. The examples are largely for self-study; although we won't go through their code in detail, they all use concepts we've already explored in this chapter.

Using Properties to Validate

Our first coding in Example 38-19 uses properties to manage three attributes. As usual, we could use simple methods instead of managed attributes, but properties help if we have already been using attributes in existing code. Properties run code automatically on attribute access but are focused on a specific set of attributes; they cannot be used to intercept all attributes generically.

To understand this code, it's crucial to notice that the attribute assignments inside the `__init__` constructor method trigger property setter methods too. When this method assigns to `self.name`, for example, it automatically invokes the `setName` method, which transforms the value and assigns it to an instance attribute called `__name` so it won't clash with the property's name.

This renaming, sometimes called *name mangling*, is important because properties use common instance state and have none of their own. Data is stored in an attribute called `__name`, and the attribute called `name` is always a property, not data. As we saw in Chapter 31, names like `__name` are known as *pseudo-private* attributes and are changed by Python to include the enclosing class's name when stored in the instance's namespace; here, this helps keep the implementation-specific attributes distinct from others, including that of the property that manages them.

In the end, this class manages attributes called `name`, `age`, and `acct`; allows the attribute `addr` to be accessed directly; and provides a read-only attribute called `remain` that is entirely virtual and computed on demand. For comparison purposes, this property-based coding weighs in at 39 lines of code (including blank lines).

Example 38-19. validate_properties.py

```
class CardHolder:
    acctlen = 8                             # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                    # Instance data
        self.name = name                    # These trigger prop setters too!
        self.age  = age                     # __X mangled to have class name
        self.addr = addr                    # addr is not managed
                                            # remain has no data
    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)       # Or @ decorators for both

    def getAge(self):
        return self.__age
    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
            self.__age = value
    age = property(getAge, setAge)

    def getAcct(self):
        return self.__acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('invalid acct number')
        else:
            self.__acct = value
    acct = property(getAcct, setAcct)

    def remainGet(self):                      # Could be a method, not attr
        return self.retireage - self.age      # Unless already using as attr
    remain = property(remainGet)
```

Testing code

To test our class, run the script in Example 38-20 in a console with the name of the class's module (sans ".py") as a single command-line argument (you could also import the class in a REPL, but we're trying to avoid repeating code here). We'll use this same test script for all four versions of this example so their output will be the same. When it runs, it makes two instances of our managed-attribute class and fetches and changes their various attributes. Operations expected to fail are wrapped in `try` statements.

Example 38-20. validate_tester.py

```
def loadclass():
    import sys, importlib
    modulename = sys.argv[1]                         # Module name in command line
    module = importlib.import_module(modulename)     # Import module by name string
    print(f'[Using: {module.CardHolder}]')           # No need for getattr() here
    return module.CardHolder

def printholder(who):
    print(who.acct, who.name, who.age, who.remain, who.addr, sep=' / ')

if __name__ == '__main__':
    CardHolder = loadclass()
    bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
    printholder(bob)
    bob.name = 'Bob Q. Smith'
    bob.age  = 50
    bob.acct = '23-45-67-89'
    printholder(bob)

    sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
    printholder(sue)
    try:
        sue.age = 200
    except: print('Bad age for Sue')

    try:
        sue.remain = 5
    except: print("Can't set sue.remain")

    try:
        sue.acct = '1234567'
    except: print('Bad acct for Sue')
```

Following is the output of our test script's code; again, this is the same for the other versions of this example ahead, except for the tested class's name. Trace through this code to see how the class's methods are invoked. Accounts are displayed with some digits hidden, names are converted to a standard format, and time remaining until retirement (hypothetically speaking) is computed when fetched using a class-attribute cutoff:

```
$ python3 validate_tester.py validate_properties
[Using: <class 'validate_properties.CardHolder'>]
12345*** / bob_smith / 40 / 22.5 / 123 main st
23456*** / bob_q._smith / 50 / 12.5 / 123 main st
56781*** / sue_jones / 35 / 27.5 / 124 main st
Bad age for Sue
Can't set sue.remain
Bad acct for Sue
```

Using Descriptors to Validate

Now, let's recode our example using *descriptors* instead of properties. As we've seen, descriptors are very similar to properties in terms of functionality and roles; in fact, properties are basically a focused form of descriptor. Like properties, descriptors are designed to handle specific attributes, not generic attribute access. Unlike properties, descriptors can also have their own state, and so are perhaps a more general scheme.

Option 1: Validating with shared descriptor-instance state (badly!)

To understand the code in Example 38-21, it's again important to notice that the attribute assignments inside the `__init__` constructor method trigger descriptor `__set__` methods. When the constructor method assigns to `self.name`, for example, it automatically invokes the `Name.__set__()` method, which transforms the value and assigns it to a descriptor attribute called `name`.

In the end, this class implements the same attributes as the prior version: it manages attributes called `name`, `age`, and `acct`; allows the attribute `addr` to be accessed directly; and provides a read-only attribute called `remain` that is entirely virtual and computed on demand. Notice how we must catch assignments to the `remain` name in its descriptor and raise an exception; as we learned earlier, if we did not do this, assigning to this attribute of an instance would silently create an instance attribute that hides the class-attribute descriptor.

For comparison purposes, this descriptor-based coding takes 45 lines of code.

Example 38-21. validate_descriptors1.py

```
class CardHolder:                                # Using shared descriptor state
    acctlen = 8                                  # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                         # Instance data
        self.name = name                         # These trigger __set__ calls too!
        self.age  = age                          # __X not needed: in descriptor
        self.addr = addr                         # addr is not managed
                                                 # remain has no data
    class Name:
        def __get__(self, instance, owner):      # Class names: CardHolder locals
            return self.name
        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            self.name = value
    name = Name()
    class Age:
        def __get__(self, instance, owner):
            return self.age                           # Use descriptor data
        def __set__(self, instance, value):
            if value < 0 or value > 150:
                raise ValueError('invalid age')
            else:
                self.age = value
    age = Age()

    class Acct:
        def __get__(self, instance, owner):
            return self.acct[:-3] + '***'
        def __set__(self, instance, value):
            value = value.replace('-', '')
            if len(value) != instance.acctlen:        # Use instance class data
                raise TypeError('invalid acct number')
            else:
                self.acct = value
    acct = Acct()

    class Remain:
        def __get__(self, instance, owner):
            return instance.retireage - instance.age    # Triggers Age.__get__
        def __set__(self, instance, value):
            raise TypeError('cannot set remain')       # Else set allowed here
    remain = Remain()
```

When run with the prior testing script, all examples in this section produce the same output as shown for properties earlier, except that the name of the class in the first line varies:

```
$ python3 validate_tester.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
…rest is same output as properties…
```

Option 2: Validating with per-client-instance state (correctly)

Unlike in the prior property-based variant, though, in Example 38-21, the actual `name` value is attached to the *descriptor* object, not the client class instance. Although we could store this value in either instance or descriptor state, the latter avoids the need to mangle names with underscores to avoid collisions. In the `CardHolder` client class, the attribute called `name` is always a descriptor object, not data.

Importantly, the downside of this scheme is that state stored inside a descriptor itself is class-level data that is effectively *shared* by all client-class instances and so cannot vary between them. That is, storing state in the *descriptor* instance instead of the *owner* (client) class instance means that the state will be the same in all owner-class instances. Descriptor state can vary only per attribute appearance.

To see this at work, try printing attributes of the `bob` instance after creating the second instance, `sue`, with the new test script in Example 38-22. The values of `sue`'s managed attributes (`name`, `age`, and `acct`) *overwrite* those of the earlier object `bob`, because both share the same, single descriptor instance attached to their class.

Example 38-22. validate_tester_plus.py

```
from validate_tester import loadclass
CardHolder = loadclass()

bob = CardHolder('1234-5678',  'Bob Smith', 40, '123 main st')
print('bob:', bob.name, bob.acct, bob.age, bob.addr)

sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
print('sue:', sue.name, sue.acct, sue.age, sue.addr)    # addr differs: client data
print('bob:', bob.name, bob.acct, bob.age, bob.addr)    # name,acct,age overwritten?
```

When this script is run with the descriptor-state `CardHolder` of Example 38-21, the results confirm the suspicion—in terms of managed attributes, bob has morphed into sue!

```
$ python3 validate_tester_plus.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
bob: bob_smith 12345*** 40 123 main st
sue: sue_jones 56781*** 35 124 main st
bob: sue_jones 56781*** 35 123 main st
```

This isn't an issue for properties because they have no state of their own, and there are valid uses for descriptor state. Such state might be used, for example, to manage descriptor implementation and data that spans all instances, and this example was coded this way on purpose to illustrate the technique. Moreover, the state scope implications of class versus instance attributes should be more or less a given at this point in the book.

However, in this particular use case, attributes of `CardHolder` objects are probably better stored as *per-instance* data instead of descriptor-instance data, perhaps using the same *__X* naming convention as the property-based equivalent to avoid name clashes in the instance—a more important factor this time, as the client is a different class with its own state attributes. Example 38-23 has the required changes; it doesn't change line counts (we're still at 45).

Example 38-23. validate_descriptors2.py

```
class CardHolder:                                 # Using per-client-instance state
    acctlen = 8                                   # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                          # Client instance data
        self.name = name                          # These trigger __set__ calls too!
        self.age  = age                           # __X needed: in client instance
        self.addr = addr                          # addr is not managed
                                                  # remain managed but has no data
    class Name:
        def __get__(self, instance, owner):       # Class names: CardHolder locals
            return instance.__name
        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            instance.__name = value
    name = Name()                                        # class.name vs mangled attr

    class Age:
        def __get__(self, instance, owner):
            return instance.__age                        # Use *instance* data
        def __set__(self, instance, value):
            if value < 0 or value > 150:
                raise ValueError('invalid age')
            else:
                instance.__age = value
    age = Age()                                         # class.age vs mangled attr

    class Acct:
        def __get__(self, instance, owner):
            return instance.__acct[:-3] + '***'
        def __set__(self, instance, value):
            value = value.replace('-', '')
            if len(value) != instance.acctlen:          # Use instance class data
                raise TypeError('invalid acct number')
            else:
                instance.__acct = value
    acct = Acct()                                       # class.acct vs mangled name

    class Remain:
        def __get__(self, instance, owner):
            return instance.retireage - instance.age    # Triggers Age.__get__
        def __set__(self, instance, value):
            raise TypeError('cannot set remain')        # Else set allowed here
    remain = Remain()
```

This supports per-instance data for the `name`, `age`, and `acct` managed fields as expected (`bob` remains `bob`), and other tests work as before:

```
$ python3 validate_tester_plus.py validate_descriptors2
[Using: <class 'validate_descriptors2.CardHolder'>]
bob: bob_smith 12345*** 40 123 main st
sue: sue_jones 56781*** 35 124 main st
bob: bob_smith 12345*** 40 123 main st

$ python3 validate_tester.py validate_descriptors2
...same output as properties, except class name...
```

One small caveat here: as coded, this version doesn't support *through-class* descriptor access because such access passes a `None` to the instance argument (also notice the attribute `__X` name mangling to `_Name__name` in the error message when the fetch attempt is made):

```
>>> from validate_descriptors1 import CardHolder
>>> pat = CardHolder('1234-5678', 'Pat Smith', 40, '123 main st')
>>> pat.name
'pat_smith'
>>> CardHolder.name
'pat_smith'

>>> from validate_descriptors2 import CardHolder
>>> pat = CardHolder('1234-5678', 'Pat Smith', 40, '123 main st')
>>> pat.name
'pat_smith'
>>> CardHolder.name
AttributeError: 'NoneType' object has no attribute '_Name__name'
```

We could detect this with a minor amount of additional code to trigger the error more explicitly, but there's probably no point—because this version stores data in the *client instance*, there's no meaning to its descriptors unless they're accompanied by a client instance (much like a normal nonbound instance method). In fact, that's really the entire point of this version's change!

Because they are classes, descriptors are a useful and powerful tool, but they present choices that can deeply impact a program's behavior. As always in OOP, choose your state retention policies carefully.

Using __getattr__ to Validate

As we've seen, the `__getattr__` method intercepts all undefined attributes, so it can be more generic than using properties or descriptors. For our example, we simply test the attribute name to know when a managed attribute is being fetched; others are stored physically on the instance and so never reach `__getattr__`. Although this approach is more general than using properties or descriptors, extra work may be required to imitate the specific attribute focus of other tools. We need to check names at runtime—a multiple-choice that's a prime role for the `match` statement—and we must code a `__setattr__` in order to intercept and validate attribute assignments.

Example 38-24 hosts the `__getattr__` version of our validations code. In the end, this class, like the prior two, manages attributes called `name`, `age`, and `acct`; allows the attribute `addr` to be accessed directly; and provides a read-only attribute called `remain` that is entirely virtual and is computed on demand.

As for the property and descriptor versions of this example, it's critical to notice that the attribute assignments inside the `__init__` constructor method trigger the class's `__setattr__` method too. When this method assigns to `self.name`, for example, it automatically invokes the `__setattr__` method, which transforms the value and assigns it to an instance attribute called `name`. By storing `name` on the instance, it ensures that future accesses will not trigger `__getattr__`. In contrast, `acct` is stored as `_acct` so that later accesses to `acct` do invoke `__getattr__`.

For comparison purposes, this alternative comes in at 34 lines of code—5 fewer than the property-based version and 11 fewer than the version using descriptors (though replacing `if` with `match` here added two lines, along with extra indentation). Clarity matters more than code size, of course, but extra code can imply extra development and maintenance work. Probably more important here are *roles*: generic tools like `__getattr__` are better suited to generic delegation, while properties and descriptors are designed to manage specific attributes.

Also note again that the code here incurs *extra calls* when setting unmanaged attributes (e.g., `addr`), although no extra calls are incurred for fetching unmanaged attributes since they are defined. Though this will likely result in negligible overhead for most programs, the more narrowly focused properties and descriptors incur an extra call only when managed attributes are accessed, and also appear in `dir` results automatically when needed by generic tools.

Example 38-24. validate_getattr.py

```
class CardHolder:
    acctlen = 8                                 # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Instance data
        self.name = name                        # These trigger __setattr__ too
        self.age  = age                         # _acct not mangled: name tested
        self.addr = addr                        # addr is not managed
                                                # remain has no data
    def __getattr__(self, name):
        match name:
            case 'acct':                              # On undefined attr fetches
                return self._acct[:-3] + '***'        # name, age, addr are defined
            case 'remain':
                return self.retireage - self.age      # Doesn't trigger __getattr__
            case _:
```

```
            raise AttributeError(name)

    def __setattr__(self, name, value):
        match name:
            case 'name':                             # On all attr assignments
                value = value.lower().replace(' ', '_')  # addr stored directly
            case 'age':                              # acct mangled to _acct
                if value < 0 or value > 150:
                    raise ValueError('invalid age')
            case 'acct':
                name  = '_acct'
                value = value.replace('-', '')
                if len(value) != self.acctlen:
                    raise TypeError('invalid acct number')
            case 'remain':
                raise TypeError('cannot set remain')
        self.__dict__[name] = value                  # Avoid looping (or object)
```

When this code is run with either test script, it produces the same output (with a different class name):

```
$ python3 validate_tester.py validate_getattr
…same output as properties, except class name…

$ python3 validate_tester_plus.py validate_getattr
…same output as instance-state descriptors, except class name…
```

Using __getattribute__ to Validate

Our final variant uses the `__getattribute__` catchall to intercept attribute fetches and manage them as needed. Every attribute fetch is caught here, so we test the attribute names to detect managed attributes and route all others to the superclass for normal fetch processing. This version uses the same `__setattr__` to catch assignments as the prior version (there is no corresponding "`__setattribute__`" in Python—so far?).

Example 38-25 codes this last mod. It works very much like the `__getattr__` version, so we won't repeat the full description here. Note, though, that because *every* attribute fetch is routed to `__getattribute__`, we don't need to mangle names to intercept them here (`acct` is stored as `acct`). On the other hand, this code must take care to route nonmanaged attribute fetches to a superclass to avoid looping or extra calls.

Also, notice that this version incurs extra calls for both setting and fetching unmanaged attributes (e.g., `addr`); if speed is paramount, this alternative may be the slowest of the bunch. For comparison purposes, this version amounts to 34 lines of code, just like the prior version (and again including 2 lines added by `match`).

Example 38-25. validate_getattribute.py

```
class CardHolder:
    acctlen = 8                              # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                     # Instance data
        self.name = name                     # These trigger __setattr__ too
        self.age  = age                      # acct not mangled: name tested
        self.addr = addr                     # addr is not managed
                                             # remain has no data
    def __getattribute__(self, name):
        superget = object.__getattribute__                # Don't loop: level up
        match name:
            case 'acct':                                  # On all attr fetches
                return superget(self, 'acct')[:-3] + '***'
            case 'remain':
                return superget(self, 'retireage') - superget(self, 'age')
            case _:
                return superget(self, name)               # name, age, addr: stored

    def __setattr__(self, name, value):
        match name:
            case 'name':                                  # On all attr assignments
                value = value.lower().replace(' ', '_')   # addr stored directly
            case 'age':
                if value < 0 or value > 150:
                    raise ValueError('invalid age')
            case 'acct':
                value = value.replace('-', '')
                if len(value) != self.acctlen:
                    raise TypeError('invalid acct number')
            case 'remain':
                raise TypeError('cannot set remain')
        self.__dict__[name] = value                        # Avoid loop, orig names
```

Both the `__getattr__` and `__getattribute__` scripts work the same as the property and per-client-instance descriptor versions when run by both tester scripts—*four ways to achieve the same goal in Python*, though they vary in structure and are perhaps less redundant in some other roles:

```
$ python3 validate_tester.py validate_getattribute
…same output as properties, except class name…

$ python3 validate_tester_plus.py validate_getattribute
…same output as instance-state descriptors, except class name…
```

Be sure to study and run this section's code on your own for more pointers on managed-attribute coding techniques.

Chapter Summary

This chapter covered the various techniques for managing access to attributes in Python, including the `__getattr__` and `__getattribute__` operator-overloading methods, and class properties and descriptors. Along the way, it compared and contrasted these tools and presented a handful of use cases to demonstrate their behavior.

Chapter 39 continues our tool-building focus with a survey of *decorators*—code run automatically at function and class creation time rather than on attribute access. Before we continue, though, let's work through a set of questions to review what we've covered here.

Test Your Knowledge: Quiz

1. How do `__getattr__` and `__getattribute__` differ?
2. How do properties and descriptors differ?
3. How are properties and decorators related?

4. What are the main functional differences between `__getattr__` and `__getattribute__` and properties and descriptors?

Test Your Knowledge: Answers

1. The `__getattr__` method is run for explicit fetches of *undefined* attributes only (i.e., those not present on an instance and not inherited from any of its classes). By contrast, the `__getattribute__` method is called for *every* explicit attribute fetch, whether the attribute is defined or not. Because of this, code inside a `__getattr__` can freely fetch other attributes if they are defined, whereas `__getattribute__` must use special code for all such attribute fetches to avoid looping or extra calls (it must route fetches to a superclass to skip itself). Neither method is run for *implicit* fetches of built-in operations (sans the next chapter's heroics).
2. Properties serve a specific role, whereas descriptors are more general. Properties define get, set, and delete functions for a specific attribute; descriptors provide a class with methods for these actions, too, but they provide extra flexibility to support more arbitrary actions. In fact, properties are really a simple way to create a specific kind of descriptor—one that runs functions on attribute accesses. Coding differs too: a property is created with a built-in function, and a descriptor is coded with a class; thus, descriptors can leverage all the usual OOP features of classes, such as inheritance. Moreover, in addition to the instance's state information, descriptors have local state of their own, which can sometimes avoid name collisions in the instance.
3. Properties can be coded with decorator syntax. Because the `property` built-in accepts a single function argument and returns a function, it can be used directly as a function decorator to define a fetch-access property. Due to the name rebinding behavior of decorators, the name of the decorated function is assigned to a property whose get accessor is set to the original function decorated (`name=property(name)`). Property `setter` and `deleter` attributes allow us to further add set and delete accessors with decoration syntax—they set the accessor to the decorated function and return the augmented property. Some may find this a bit clumsy, but this is subjective.
4. The `__getattr__` and `__getattribute__` methods are more generic: they can be used to catch arbitrarily many attributes. In contrast, each property or descriptor provides access interception for only one *specific* attribute—we can't catch every attribute fetch with a single property or descriptor. On the other hand, properties and descriptors handle both attribute fetch and *assignment* by design: `__getattr__` and `__getattribute__` handle fetches only; to intercept assignments as well, `__setattr__` must also be coded. The implementation is also different: `__getattr__` and `__getattribute__` are operator-overloading methods, whereas properties and descriptors are objects manually assigned to class attributes. Unlike the others, properties and descriptors can also sometimes avoid extra calls on assignment to unmanaged names and show up in `dir` results automatically, but are also narrower in scope—they can't address generic delegation goals. In Python evolution, new features tend to offer alternatives but often do not fully subsume what came before.