Chapter 28. A More Realistic Example

We'll dig into more class syntax details in the next chapter. Before we do, though, let's take a short detour to explore a realistic example of classes in action that's more practical than what we've seen so far. In this chapter, we're going to build a set of classes that do something concrete—recording and processing information about people. As you'll see, what we call *instances* and *classes* in Python programming can often serve the same roles as *records* and *programs* in more traditional terms. The main difference here is the customization that inheritance will enable.

Specifically, in this chapter we're going to code two classes:

- `Person` —a class that creates and processes information about people
- `Manager` —a customization of `Person` that modifies inherited behavior

Along the way, we'll make instances of both classes and test out their functionality. When we're done, this chapter will also show you a nice example use case for classes—we'll store our instances in a simple object-oriented database, to make them permanent. That way, you can use this code as a template for fleshing out a full-blown personal database of your own written entirely in Python.

Besides actual utility, though, our aim here is also *educational*: this chapter provides a tutorial on object-oriented programming in Python. Often, people grasp the last chapter's class syntax in the abstract but have trouble seeing how to get started when confronted with coding a new class from scratch. Toward this end, we'll take it one step at a time here, to help you learn the basics; we'll build up the classes gradually, so you can see how their features come together in complete programs.

In the end, our classes will still be relatively small in terms of code, but they will demonstrate *all* of the main ideas in Python's OOP model. Despite its syntax details, Python's class system really is largely just a matter of searching for an attribute in a tree of objects, along with a special first argument for functions.

Step 1: Making Instances

OK, so much for the design phase—let's move on to implementation. Our first task is to start coding the main class, `Person`. In your favorite text editor, open a new file for the code we'll be writing.

As noted in the prior chapter, it's a fairly strong convention in Python to begin module names with a lowercase letter and class names with an uppercase letter. Like the name of `self` arguments in methods, this is not required by the language, but it's so common that deviating might be confusing to people who later read your code. To conform, we'll call our new module file *person.py* and our class within it `Person`, as in Example 28-1.

Example 28-1. person_1.py (start)

```
class Person:                                # Start a cla
```

We're going to change this file as we go. To help you keep track of its variations, we'll append an *example number* to the filename in captions, and use that number both in the examples package and in launches and imports here. Changes will also be shown in bold font on each revision. The intent is to show mods to a single file, but books are linear.

As noted in the prior chapter, we can code any number of functions and classes in a single module file in Python, and this one's *person.py* name might not make much sense if we add unrelated components to it later. For now, we'll assume everything in it will be `Person`-related. It probably should be anyhow—as we've learned, modules tend to work best when they have a single, *cohesive* purpose.

Coding Constructors

Now, the first thing we want to do with our `Person` class is record basic information about people—to fill out record fields, if you will. Of course, these are known as instance object *attributes* in Python-speak, and they generally are created by assignment to `self` attributes in a class's method functions. The normal way to give instance attributes their first values is to assign them to `self` in the `__init__` *constructor method*, which contains

code run automatically by Python each time an instance is created. Let's add one to our class, in Example 28-2.

Example 28-2. person_2.py (add attribute initialization)

```
class Person:
    def __init__(self, name, job, pay):      # Construc
        self.name = name                      # Fill out
        self.job  = job                       # self is
        self.pay  = pay
```

This is a very common coding pattern: we pass in the data to be attached to an instance as arguments to the constructor method and assign them to `self` to retain them permanently. In OO terms, `self` is the newly created instance object, and `name`, `job`, and `pay` become *state information*—descriptive data saved on an object for later use. Although other techniques (such as enclosing scope reference closures) can save details, too, instance attributes make this very explicit and easy to understand.

Notice that the argument names appear *twice* here. This code might even seem a bit redundant at first, but it's not. The `job` argument, for example, is a local variable in the scope of the `__init__` function, but `self.job` is an attribute of the instance that's the implied subject of the method call. They are two different variables, which happen to have the same name. By assigning the `job` local to the `self.job` attribute with `self.job=job`, we save the passed-in `job` on the instance for later use. As usual in Python, where a name is assigned, or what object it is assigned to, determines what it means.

Speaking of arguments, there's really nothing magical about `__init__`, apart from the fact that it's called automatically when an instance is made, and has a special first argument. Despite its weird name, it's a normal function and supports all the features of functions we've already covered. We can, for example, provide *defaults* for some of its arguments, so they need not be provided in cases where their values aren't available or useful.

To demonstrate, let's make the `job` argument optional—it will default to `None`, meaning the person being created is not (currently?) employed. If `job` defaults to `None`, we'll probably want to default `pay` to `0`, too, for consistency (unless some of the people you know manage to get paid without having jobs). In fact, we have to specify a default for `pay` because according

to Python's syntax rules and Chapter 18, any arguments in a function's header after the first default must all have defaults, too. Example 28-3 codes the mod.

Example 28-3. person_3.py (add constructor defaults)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
```

What this code means is that we'll need to pass in a name when making `Person` s, but `job` and `pay` are now optional; they'll default to `None` and `0` if omitted. The `self` argument, as usual, is filled in by Python automatically to refer to the instance object—assigning values to attributes of `self` attaches them to the new instance.

Testing as You Go

This class doesn't do much yet—it essentially just fills out the fields of a new record—but it's a real working class. At this point, we could add more code to it for more features, but we won't do that yet. As you've probably begun to appreciate already, programming in Python is really a matter of *incremental prototyping*—you write some code, test it, write more code, test again, and so on. Because Python provides both an interactive session and nearly immediate turnaround after code changes, it's more natural to test as you go than to write a huge amount of code to test all at once.

Before adding more features, then, let's test what we've got so far by making a few instances of our class and displaying their attributes as created by the constructor. We could do this interactively, but as you've also probably surmised by now, interactive testing has its limits—it gets tedious to have to reimport modules and retype test cases each time you start a new testing session. More commonly, Python programmers use the interactive prompt for simple one-off tests but do more substantial testing by writing code at the bottom of the file that contains the objects to be tested, as in Example 28-4.

Example 28-4. person_4.py (add incremental self-test code)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

bob = Person('Bob Smith')                          # Tes
sue = Person('Sue Jones', job='dev', pay=100000)   # Rur
print(bob.name, bob.pay)                           # Fet
print(sue.name, sue.pay)                           # sue
```

Notice here that the `bob` object accepts the defaults for `job` and `pay`, but `sue` provides values explicitly. Also, note how we use *keyword arguments* when making `sue`; we could pass by position instead, but the keywords may help remind us later what the data is, and they allow us to pass the arguments in any left-to-right order we like. Again, despite its unusual name, `__init__` is a normal function, supporting everything you already know about functions—including both defaults and pass-by-name keyword arguments.

When this file runs as a script, the test code at the bottom makes two instances of our class and prints two attributes of each— `name` and `pay`:

```
$ python3 person_4.py
Bob Smith 0
Sue Jones 100000
```

You can also type this file's test code at Python's interactive prompt (assuming you import the `Person` class there first), but coding canned tests inside the module file like this makes it much easier to rerun them in the future.

Although this is fairly simple code, it's already demonstrating something important. Notice that `bob`'s `name` is not `sue`'s, and `sue`'s `pay` is not `bob`'s. Each is an independent record of information. Technically, `bob` and `sue` are both *namespace objects*—like all class instances, they each have their own independent copy of the state information created by the class. Because each instance of a class has its own set of `self` attributes, classes

are a natural for recording information for multiple objects this way; just like built-in types such as lists and dictionaries, classes serve as a sort of *object factory*.

Other Python program structures, such as functions and modules, have no such concept. Chapter 17's closure functions come close in terms of per-call state but don't have the multiple methods, inheritance, and larger structure we get from classes.

Using Code Two Ways

As is, the test code at the bottom of the file works, but there's a big catch—its top-level `print` statements run both when the file is run as a script and when it is imported as a module. This means if we ever decide to import the class in this file in order to use it somewhere else (and we will soon in this chapter), we'll see the output of its test code every time the file is imported. That's not very good software citizenship, though: client programs probably don't care about our internal tests and won't want to see our output mixed in with their own.

Although we could split the test code off into a separate file, it's often more convenient to code tests in the same file as the items to be tested. It would be better to arrange to run the test statements at the bottom *only* when the file is run for testing, not when the file is imported. As linear readers of this book have already learned, that's exactly what the module `__name__` check is designed for. Example 28-5 shows what this addition looks like—simply add the required test and indent your self-test code.

Example 28-5. person_5.py (support both imports and run/tests)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                  # When run
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
```

```
print(bob.name, bob.pay)
print(sue.name, sue.pay)
```

Now, we get exactly the behavior we're after—running the file as a top-level script tests it because its `__name__` is `__main__`, but importing it as a library of classes later does not:

```
$ python3 person_5.py
Bob Smith 0
Sue Jones 100000

$ python3
>>> import person_5
>>>
```

When imported, the file now defines the class but does not use it. When run directly, this file creates two instances of our class as before, and prints two attributes of each; again, because each instance is an independent namespace object, the values of their attributes differ.

Step 2: Adding Behavior Methods

Everything looks good so far—at this point, our class is essentially a record *factory*; it creates and fills out fields of records (attributes of instances, in Pythonic terms). Even as limited as it is, though, we can still run some operations on its objects. Although classes add an extra layer of structure, they ultimately do most of their work by embedding and processing *core object types* like lists and strings. In other words, if you already know how to use Python's simple core objects, you already know much of the Python class story; classes are really just a minor structural extension.

For example, the `name` field of our objects is a simple string, so we can extract last names from our objects by splitting on spaces and indexing. These are all core-object operations, which work whether their subjects are embedded in class instances or not:

```
>>> name = 'Bob Smith'      # Simple string, outside cl
>>> name.split()            # Extract last name
['Bob', 'Smith']
```

```
>>> name.split()[-1]        # Or [1], if always just tw
'Smith'
```

Similarly, we can give an object a pay raise by updating its `pay` field—that is, by changing its state information in place with an assignment. This task also involves basic operations that work on Python's core objects, regardless of whether they are standalone or embedded in a class structure (formatting here masks any extraneous digits):

```
>>> pay = 100000            # Simple variable, outside
>>> pay *= 1.10             # Give a 10% raise
>>> print(f'{pay:,.2f}')    # Or: pay = pay * 1.10, if
110,000.00                  # Or: pay = pay + (pay * .1
```

To apply these operations to the `Person` objects created by our script, simply do to `bob.name` and `sue.pay` what we just did to `name` and `pay`, as listed in Example 28-6. The operations are the same, but the subjects are attached as attributes to objects created from our class.

Example 28-6. person_6.py (process embedded built-in objects)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])            # Extract ob
    sue.pay *= 1.10                        # Give this
    print(f'{sue.pay:,.2f}')
```

We've added the last three lines here; when they're run, we extract `bob`'s last name by using basic string and list operations on his name field, and give `sue` a pay raise by modifying her `pay` attribute in place with basic number operations. In a sense, `sue` is also a *mutable* object—her state changes in

place just like a list after an `append` call. Here's the new version's output when run as a top-level script:

```
$ python3 person_6.py
Bob Smith 0
Sue Jones 100000
Smith
110,000.00
```

The preceding code works as planned, but if you show it to a veteran software developer he or she will probably tell you that its general approach is not a great idea in practice. Hardcoding operations like these *outside* of the class can lead to maintenance problems in the future.

For example, what if you've hardcoded the last-name-extraction formula at many different places in your program? If you ever need to change the way it works (to support a new name structure, for instance), you'll need to hunt down and update *every* occurrence. Similarly, if the pay-raise code ever changes (e.g., to require approval or database updates), you may have multiple copies to modify. Just finding all the appearances of such code may be problematic in larger programs—they may be scattered across many files and folders, split into individual steps, and so on. In a prototype like this, frequent change is almost guaranteed.

Coding Methods

What we really want to do here is employ a software design concept known as *encapsulation*—wrapping up operation logic behind interfaces, such that each operation is coded only once in our program. That way, if our needs change in the future, there is just one copy to update. Moreover, we're free to change the single copy's internals almost arbitrarily, without breaking the code that uses it.

In Python terms, we want to code operations on objects in a class's *methods*, instead of littering them throughout our program. In fact, this is one of the things that classes are very good at—*factoring* code to remove *redundancy* and thus optimize maintainability. As an added bonus, turning operations into methods enables them to be applied to any instance of the class, not just those that they've been hardcoded to process.

This is all simpler in code than it may sound in theory. Example 28-7 achieves encapsulation by moving the two operations from code outside the class to methods inside the class. While we're at it, let's change our self-test code at the bottom to use the new methods we're creating, instead of hardcoding operations.

Example 28-7. person_7.py (add methods to encapsulate operations)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                  #
        return self.name.split()[-1]                     #
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))         #

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())                #
    sue.giveRaise(.10)                                   #
    print(sue.pay)
```

As we've learned, *methods* are simply normal functions that are attached to classes and designed to process instances of those classes. The instance is the subject of the method call and is passed to the method's `self` argument automatically.

The transformation to the methods in this version is straightforward. The new `lastName` method, for example, simply does for `self` what the previous version hardcoded for `bob`, because `self` is the implied subject when the method is called. `lastName` also returns the result because this operation is a called function now; it computes a value for its caller to use arbitrarily, even if it is just to be printed. Similarly, the new `giveRaise` method just does for `self` what we did for `sue` before.

When run now, our file's output is similar to before—we've mostly just *refactored* the code to allow for easier changes in the future (command lines that run the most recent mod are often omitted from here on for space):

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

A few coding details are worth pointing out here. First, notice that `sue` 's pay is now still an *integer* after a pay raise—we convert the math result back to an integer by calling the `int` built-in within the method. Changing the value to either `int` or `float` is probably not a significant concern for this demo: integer and floating-point objects have the same interfaces and can be mixed within expressions. Still, we may need to address truncation and rounding issues in a real system—money probably is significant to `Person` s!

As covered in Chapter 5, we might handle this by using the `round(N, 2)` built-in to round and retain cents, using the `decimal` type to fix precision, or storing monetary values as full floating-point numbers and displaying them with a formatting string to show cents as we did earlier. For now, we'll simply truncate any cents with `int` .

Second, notice that we're also printing `sue` 's last name this time—because the last-name logic has been encapsulated in a method, we get to use it on *any instance* of the class. As we've seen, Python tells a method which instance to process by automatically passing it in to the first argument, usually called `self` . Specifically:

- In the first call, `bob.lastName()` , `bob` is the implied subject passed to `self` .
- In the second call, `sue.lastName()` , `sue` goes to `self` instead.

Trace through these calls to see how the instance winds up in `self` —it's a key concept. The net effect is that the method fetches the name of the implied subject each time. The same happens for `giveRaise` . We could, for example, give `bob` a raise by calling `giveRaise` for both instances this way, too. Unfortunately for `bob` , though, his zero starting pay will prevent him from getting a raise as the program is currently coded—nothing times

anything is nothing, something we may want to address in a future 2.0 release of our software.

Finally, notice that the `giveRaise` method assumes that `percent` is passed in as a floating-point number between zero and one. That may be too radical an assumption in the real world (a 1,000% raise would probably be a bug for most of us!); we'll let it pass for this prototype, but we might want to test or at least document this in a future iteration of this code. Stay tuned for a rehash of this idea in later chapters, where we'll explore *function decorators* and Python's `assert` statement as alternatives that can do the validity test for us automatically during development. In Chapter 39, for example, we'll write a tool that lets us validate with strange incantations like the following:

```
    @rangetest(percent=(0.0, 1.0))                # Use
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Step 3: Operator Overloading

At this point, we have a fairly full-featured class that generates and initializes instances, along with two new bits of behavior for processing instances in the form of methods. So far, so good.

As it stands, though, testing is still a bit less convenient than it needs to be—to trace our objects, we have to manually fetch and print *individual attributes* (e.g., `bob.name` , `sue.pay`). It would be nice if displaying an instance all at once actually gave us some useful information. Unfortunately, the default display format for an instance object isn't very good—it displays the object's class name and its address in memory (which is essentially useless in Python, except as a unique identifier).

To see this, change the last line in the latest version of our script to `print(sue)` so it displays the object as a whole. Here's what you'll get—as coded, the output says that `sue` is an object, but not much more:

```
Bob Smith 0
Sue Jones 100000
```

```
Smith Jones
<__main__.Person object at 0x104972630>
```

Providing Print Displays

Fortunately, it's easy to do better by employing *operator overloading*—coding methods in a class that intercept and process built-in operations when run on the class's instances. Specifically, we can make use of what are probably the second most commonly used operator-overloading methods in Python, after `__init__` : the `__repr__` method we'll deploy here, and its `__str__` twin introduced in the preceding chapter.

These methods are run automatically every time an instance is converted to its print string. Because that's what `print` normally does, the net transitive effect is that printing an object displays whatever is returned by the object's `__str__` or `__repr__` method, if the object either defines one itself or inherits one from a superclass. Double-underscored names are inherited just like any other.

Technically, `__str__` is preferred by `print` and `str` , and `__repr__` is used both as a fallback for these, as well as by interactive echoes and nested appearances. Although the two can be used to implement different displays in different contexts (e.g., user- or developer-focused), coding `__repr__` alone suffices to give a single display in all cases. This allows clients to provide an alternative display with `__str__` , though this is a moot point for this demo.

The `__init__` constructor method we've already coded is, strictly speaking, operator overloading too—it is run automatically at construction time to initialize a newly created instance. Constructors are so common, though, that they almost seem like a special case. More focused methods like `__repr__` allow us to tap into specific operations and provide *specialized behavior* when our objects are used in those contexts.

Let's put this into code. Example 28-8 extends our class to give a custom display that lists attributes when our class's instances are displayed as a whole, instead of relying on the less useful default display.

Example 28-8. person_8.py (add __repr__ overload method for displays)

```
class Person:
    def __init__(self, name, job=None, pay=0):
```

```
            self.name = name
            self.job  = job
            self.pay  = pay
        def lastName(self):
            return self.name.split()[-1]
        def giveRaise(self, percent):
            self.pay = int(self.pay * (1 + percent))
        def __repr__(self):
            return f'[Person: {self.name} ${self.pay:,}]'

    if __name__ == '__main__':
        bob = Person('Bob Smith')
        sue = Person('Sue Jones', job='dev', pay=100000)
        print(bob)
        print(sue)
        print(bob.lastName(), sue.lastName())
        sue.giveRaise(.10)
        print(sue)
```

Notice that we're doing f-string formatting to build the display string in `__repr__` here, with comma separators for pay; at the bottom, classes use built-in operations like these to get their work done. Again, everything you've already learned about both built-in objects and user-defined functions applies to class-based code. Classes largely just add an additional layer of *structure* that packages functions and data together and supports extensions.

We've also changed our self-test code to print objects directly, instead of printing individual attributes. When run, the output is more uniform now; the "[…]" lines are returned by `__repr__`, run automatically by print operations:

```
[Person: Bob Smith $0]
[Person: Sue Jones $100,000]
Smith Jones
[Person: Sue Jones $110,000]
```

Step 4: Customizing Behavior by Subclassing

At this point, our class captures much of the OOP machinery in Python: it makes instances, provides behavior in methods, and even does a bit of operator overloading now to intercept print operations in `__repr__`. It effectively packages our data and logic together into a single, self-contained *software component*, making it easy to locate code and straightforward to change it in the future. By allowing us to encapsulate behavior, it also allows us to factor that code to avoid redundancy and its associated maintenance headaches.

The only major OOP concept it does not yet capture is *customization by inheritance*. In some sense, we're already doing inheritance, because instances inherit methods from their classes. To demonstrate the real power of OOP, though, we need to define a superclass/subclass relationship that allows us to extend our software and replace bits of inherited behavior. That's the main idea behind OOP, after all; by fostering a coding model based upon customization of work already done, it can dramatically cut development time when used well.

Coding Subclasses

As a next step, then, let's put OOP's methodology to use and customize our `Person` class by extending our software hierarchy. For the purpose of this tutorial, we'll define a subclass of `Person` called `Manager` that replaces the inherited `giveRaise` method with a more specialized version. Our new class begins as follows:

```
class Manager(Person):                                  # Defir
```

This code means that we're defining a new class named `Manager`, which inherits from and may add customizations to the superclass `Person`. In plain terms, a `Manager` is almost like a `Person` … (admittedly, a very long journey for a very small joke), but `Manager` has a custom way to give raises.[1]

For the sake of argument, let's assume that when a `Manager` gets a raise, it receives the passed-in percentage as usual, but also gets an extra bonus that defaults to 10%. For instance, if a `Manager` 's raise is specified as 10%, it will really get 20%. (Any relation to `Person` s living or dead is, of course, strictly coincidental.) Our new method begins as follows; because this redefinition of `giveRaise` will be closer in the class tree to `Manager` instances than the original version in `Person` , it effectively replaces, and thereby customizes, the operation. Recall that according to the inheritance search rules, the *lowest* version of the name wins (we'll add this code to the file in a moment):

```
class Manager(Person):                          # Inher
    def giveRaise(self, percent, bonus=.10):    # Redef
```

Augmenting Methods: The Bad Way

Now, there are two ways we might code this `Manager` customization: a good way and a bad way. Let's start with the *bad way* since it might be a bit easier to understand. The bad way is to cut and paste the code of `giveRaise` in `Person` and modify it for `Manager` , like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus)
```

This would work as advertised—when we later call the `giveRaise` method of a `Manager` instance, it would run this custom version, which tacks on the extra bonus. So what's wrong with something that runs correctly?

The problem here is a very general one: anytime you copy code with cut and paste, you essentially *double* your maintenance effort in the future. Think about it: because we copied the original version, if we ever have to change the way raises are given (and we probably will), we'll have to change the code in *two* places, not one. Although this is a small and artificial example, it's also representative of a universal issue—anytime you're tempted to program by copying code this way, you probably want to look for a better approach.

Augmenting Methods: The Good Way

What we really want to do here is somehow *augment* the original `giveRaise`, instead of replacing it altogether. The *good way* to do that in Python is by calling to the original version directly, with augmented arguments, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

This code leverages the fact that a class's method can always be called either through an *instance* (the usual way, where Python sends the instance to the `self` argument automatically) or through the *class* (the less common scheme, where you must pass the instance manually). In more symbolic terms, a normal method call of this form:

```
instance.method(args…)
```

is automatically translated by Python into this equivalent form:

```
class.method(instance, args…)
```

where the class containing the method to be run is determined by the inheritance search rule applied to the method's name. You can code *either* form in your script, but there is a slight asymmetry between the two—you must remember to pass along the instance manually if you call through the class directly.

The method always needs a subject instance one way or another, and Python provides it automatically only for calls made through an instance. For calls through the class name, you need to send an instance to `self` yourself; and for code inside a method like `giveRaise`, `self` already *is* the subject of the call, and hence the instance to pass along.

Calling through the class directly effectively subverts inheritance and kicks the call higher up the class tree to run a specific version. In our case, we can use this technique to invoke the default `giveRaise` in `Person`, even

though it's been redefined at the `Manager` level. In fact, we *must* call through `Person` this way, because a `self.giveRaise()` inside `Manager`'s `giveRaise` code would loop—since `self` already is a `Manager`, `self.giveRaise()` would resolve again to `Manager`'s `giveRaise`, and so on, *recursively*, until available memory is exhausted.

Call syntax aside, this "good" version may seem like a small difference in code, but it can make a huge difference for future *code maintenance*—because the `giveRaise` logic lives in just one place now (`Person`'s method), we have only one version to change in the future as needs evolve. And really, this form captures our intent more directly anyhow—we want to perform the standard `giveRaise` operation, but simply tack on an extra bonus. Example 28-9 lists our entire module file with this step applied.

Example 28-9. person_9.py (add method customization in a subclass)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return f'[Person: {self.name} ${self.pay:,}]'

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    pat = Manager('Pat Jones', 'mgr', 50000)
    pat.giveRaise(.10)
```

```
print(pat.lastName())
print(pat)
```

To test our `Manager` subclass customization, we've also added self-test code that makes a `Manager`, calls its methods, and prints it. When we make a `Manager`, we pass in a name, and an optional job and pay as before—because `Manager` had no `__init__` constructor, it inherits that in `Person`. Here's the new version's output:

```
[Person: Bob Smith $0]
[Person: Sue Jones $100,000]
Smith Jones
[Person: Sue Jones $110,000]
Jones
[Person: Pat Jones $60,000]
```

Everything looks good here: `bob` and `sue` are as before, and when `pat` the `Manager` is given a 10% raise, he or she really gets 20% (pay goes from $50K to $60K), because the customized `giveRaise` in `Manager` is run for this person only. Also notice how printing `pat` as a whole at the end of the test code displays the nice format defined in `Person`'s `__repr__`: `Manager` objects get this, `lastName`, and the `__init__` constructor method's code "for free" from `Person`, by inheritance.

THE SUPER ALTERNATIVE

To extend inherited methods, the examples in this chapter simply call the original through the superclass name: `Person.giveRaise(…)` . This is the explicit, traditional, and simplest scheme in Python, and the one used in most of this book. It also follows the same pattern we'll code to access *class attributes*: names attached to a class and shared by all instances, introduced ahead. Methods are just callable class attributes.

Java programmers may especially be interested to know that Python also has a `super` built-in function that allows calling back to a superclass's methods more generically. In the custom `giveRaise` of Example 28-9's `Manager` class, both of the following would work the same way:

```
Person.giveRaise(self, percent + bonus)      # Explicit,
super().giveRaise(percent + bonus)           # Implicit,
```

The `super` built-in, however, relies on unusual semantics; works unevenly with Python's operator overloading; and does not always mesh well with multiple inheritance, where a single method call may not suffice. It's a special-case tool with a single role, which is redundant with general class-name references.

In its defense, the `super` call has a valid role too—cooperative same-named method dispatch in multiple inheritance trees—but this relies on the "MRO" class ordering of Chapter 31, which many find artificial; requires universal deployment to be used reliably; does not fully support method replacement and varying argument lists; and to many observers seems an esoteric solution to a role that is rare in real Python code.

Because of these downsides, this book prefers to call superclasses by explicit name instead of `super` , recommends the same policy for newcomers, and defers presenting `super` until Chapter 32. You're of course free to draw your own conclusions there, but `super` is usually best had after you learn the simpler and more explicit ways of achieving the same goals in Python, especially if you're new to OOP. Topics like cooperative multiple-inheritance dispatch are a lot to digest—for beginners and experts alike.

And to any Java programmers in the audience: try resisting the temptation to use Python's `super` until you've had a chance to study its subtle

implications. This call is benign in single-inheritance of the sort you're used to, but once you step up to Python's multiple inheritance, `super` is not what you think it is, and more than you probably expect. The class it picks can vary per context, and may not even be a superclass at all!

Polymorphism in Action

To make this acquisition of inherited behavior even more striking, add the following code at the end of our file temporarily (this is file *person_9_plus.py* in the examples package, but doesn't warrant a full listing or caption here):

```
if __name__ == '__main__':
    …
    print('--All three--')
    for obj in (bob, sue, pat):                 # Process
        obj.giveRaise(.10)                      # Run thi
        print(obj)                              # Run the
```

Here's the resulting output, showing just its new parts (and the accumulated effects of two raises per person):

```
…
--All three--
[Person: Bob Smith $0]
[Person: Sue Jones $121,000]
[Person: Pat Jones $72,000]
```

In the added code, `object` is *either* a `Person` or a `Manager`, and Python runs the appropriate `giveRaise` automatically—our original version in `Person` for `bob` and `sue`, and our customized version in `Manager` for `pat`. Trace the method calls yourself to see how Python selects the right `giveRaise` method for each object.

This is just Python's notion of *polymorphism*, which we met earlier in the book, at work again—what `giveRaise` does depends on what you do it to. Here, it's made all the more obvious when it selects from code we've written ourselves in classes. The practical effect in this code is that `sue` gets another 10% but `pat` gets another 20% because `giveRaise` is dispatched based on the object's type. As we've seen, polymorphism is at the heart of Python's

flexibility. Passing any of our three objects to a function that calls a `giveRaise` method, for example, would have the same effect: the appropriate version would be run automatically, depending on which type of object was passed.

On the other hand, printing runs the *same* `__repr__` for all three objects, because it's coded just once in `Person`. `Manager` both specializes and applies the code we originally wrote in `Person`. Although this example is small, it's already leveraging OOP's talent for code customization and reuse; with classes, this almost seems automatic at times.

Inherit, Customize, and Extend

In fact, classes can be even more flexible than our example implies. In general, classes can *inherit*, *customize*, or *extend* existing code in superclasses. For example, although we're focused on customization here, we can also add unique methods to `Manager` that are not present in `Person`, if `Manager`s require something different. The following abstract snippet illustrates. Here, `giveRaise` redefines a superclass's method to customize it, but `someThingElse` defines something new to extend:

```
class Person:
    def lastName(self): …
    def giveRaise(self): …
    def __repr__(self): …

class Manager(Person):                  # Inherit
    def giveRaise(self, …): …           # Customize
    def someThingElse(self, …): …       # Extend

pat = Manager()
pat.lastName()           # Inherited verbatim
pat.giveRaise()          # Customized version
pat.someThingElse()      # Extension here
print(pat)               # Inherited overload method
```

Extra methods like this code's `someThingElse` augment the existing software and are available on `Manager` objects only, not on `Person`s. For the purposes of this tutorial, however, we'll limit our scope to customizing some of `Person`'s behavior by redefining it, not adding to it.

OOP: The Big Idea

As is, our code may be small, but it's fairly functional. And really, it already illustrates the main point behind OOP in general: in OOP, we program by *customizing* what has already been done, rather than copying or changing existing code. This isn't always an obvious win to newcomers at first glance, especially given the extra coding requirements of classes. But overall, the programming style implied by classes can cut development time radically compared to other approaches.

For instance, in our example we could theoretically have implemented a custom `giveRaise` operation without subclassing, but none of the other options yield code as optimal as ours:

- Although we could have simply coded `Manager` *from scratch* as new, independent code, we would have had to reimplement all the behaviors in `Person` that are the same for `Manager` s.
- Although we could have simply *changed* the existing `Person` class in place for the requirements of `Manager` 's `giveRaise` , doing so would break code that still needs the original `Person` behavior.
- Although we could have simply *copied* the `Person` class in its entirety, renamed the copy to `Manager` , and changed its `giveRaise` , doing so would introduce code redundancy that would double our work in the future—changes made to `Person` in the future would not be picked up automatically, but would have to be manually propagated to `Manager` 's code. As usual, the cut-and-paste approach may seem quick now, but it doubles your work in the future.

The *customizable hierarchies* we can build with classes provide a much better solution for software that will evolve over time. No other tools in Python support this development mode. Because we can tailor and extend our prior work by coding new subclasses, we can leverage what we've already done, rather than starting anew each time, breaking what already works, or introducing multiple redundant copies. When done right, OOP is a powerful ally.

Step 5: Customizing Constructors, Too

Our code works as it is, but if you study the current version closely, you may be struck by something a bit odd—it seems pointless to have to provide a `mgr` job name for `Manager` objects when we create them: this is already implied by the class itself. It would be better if we could somehow fill in this value automatically when a `Manager` is made.

The trick we need to improve on this turns out to be the *same* as the one we employed in the prior section: we want to customize the constructor logic for `Manager` s in such a way as to provide a job name automatically. In terms of code, we want to redefine an `__init__` method in `Manager` that provides the `mgr` string for us. And as in `giveRaise` customization, we also want to run the original `__init__` in `Person` by calling through the class name, so it still initializes our objects' state information attributes.

The mods in Example 28-10 will do the job—we've coded the new `Manager` constructor and changed the call that creates `pat` to not pass in the `mgr` job name.

Example 28-10. person_10.py (add constructor customization in a subclass)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return f'[Person: {self.name} ${self.pay:,}]'

class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
```

```
bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
print(bob)
print(sue)
print(bob.lastName(), sue.lastName())
sue.giveRaise(.10)
print(sue)
pat = Manager('Pat Jones', 50000)
pat.giveRaise(.10)
print(pat.lastName())
print(pat)
```

Again, we're using the same technique to augment the `__init__` constructor here that we used for `giveRaise` earlier—running the superclass version by calling through the class name directly and passing the `self` instance along explicitly. Although the constructor has a strange name, the effect is identical. Because we need `Person`'s construction logic to run too (to initialize instance attributes), we really have to call it this way; otherwise, instances would not have any attributes attached.

Calling superclass constructors from redefinitions this way turns out to be a very common coding pattern in Python. By itself, Python uses inheritance to look for and call only *one* `__init__` method at construction time—the *lowest* one in the class tree. If you need higher `__init__` methods to be run at construction time (and you usually do), you must call them manually, and usually through the superclass's name. The upside to this is that you can be explicit about which argument to pass up to the superclass's constructor and can choose to *not* call it at all: not calling the superclass constructor allows you to replace its logic altogether, rather than augmenting it.

The output of this file's self-test code is the same as before—we haven't changed what this example does, we've simply restructured it to get rid of some logical redundancy:

```
$ python3 person_10.py
[Person: Bob Smith $0]
[Person: Sue Jones $100,000]
Smith Jones
[Person: Sue Jones $110,000]
Jones
[Person: Pat Jones $60,000]
```

Per the sidebar "The super Alternative", an implicit `super().__init__(…)` sans `self` would run the superclass constructor too, and this is a common role for this built-in in single-inheritance class trees. For all the reasons noted earlier, though, let's stick to the explicit and general for now.

OOP Is Simpler Than You May Think

In this complete form, and despite their relatively small sizes, our classes capture nearly all the important concepts in Python's OOP machinery:

- Instance creation—filling out instance attributes
- Behavior methods—encapsulating logic in a class's methods
- Operator overloading—providing behavior for built-in operations like printing
- Customizing behavior—redefining methods in subclasses to specialize them
- Customizing constructors—adding initialization logic to superclass steps

Most of these concepts are based upon just three simple ideas: the inheritance search for attributes in object trees, the special `self` argument in methods, and operator overloading's automatic dispatch to methods.

Along the way, we've also made our code easy to change in the future, by harnessing the class's propensity for factoring code to reduce *redundancy*. For example, we wrapped up logic in methods and called back to superclass methods from extensions to avoid having multiple copies of the same code. Most of these steps were a natural outgrowth of the structuring power of classes.

By and large, that's all there is to OOP in Python. Classes certainly can become larger than this, and there are some more advanced class concepts, such as decorators and metaclasses, which we will explore in later chapters. In terms of the basics, though, our classes already do it all. In fact, if you've grasped the workings of the classes we've written, most OOP Python code should now be within your reach.

Other Ways to Combine Classes: Composites

Having said that, it should be noted that although the basic mechanics of OOP are simple in Python, some of the art in larger programs lies in the way that classes are put together. We're focusing on *inheritance* in this tutorial because that's the mechanism the Python language provides, but programmers sometimes combine classes in other ways, too.

For example, a common coding pattern involves nesting objects inside each other to build up *composites*. We'll explore this pattern in more detail in Chapter 31, which is really more about design than about Python. As a quick example, though, we could use this composition idea to code our `Manager` extension by *embedding* a `Person`, instead of inheriting from it.

The alternative `Manager` in Example 28-11 does so in part by using the `__getattr__` operator-overloading method to intercept undefined attribute fetches, and the `getattr` built-in to *delegate* them to the embedded object:

- The `__getattr__` method isn't covered in full until Chapter 30, but its usage is simple enough to employ here—it's automatically run for all fetches of undefined attributes (those not found in the instance by normal inheritance search). And yes, this is the same name used for the nonclass module-attributes hook of Chapter 25.
- The `getattr` call was introduced in Chapter 25—it's the same as `X.Y` attribute fetch notation and thus performs inheritance, but the attribute name `Y` is evaluated to yield a string, not a name interpreted literally.

More specifically here, the constructor makes the embedded object with job name; `giveRaise` customizes by changing the argument passed along to the embedded object; and `lastName` triggers `__getattr__` to reroute to the embedded object. In effect, `Manager` becomes a controller layer that passes calls *down* to the embedded object, rather than *up* to superclass methods.

Example 28-11. person-composite.py (embedding-based Manager alternative)

```
from person_10 import Person                        # E

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)      # E
```

```
        def giveRaise(self, percent, bonus=.10):
            self.person.giveRaise(percent + bonus)      # J

        def __getattr__(self, attr):
            return getattr(self.person, attr)           # D

        def __repr__(self):
            return str(self.person)                     # M

    if __name__ == '__main__':
        pat = Manager('Pat Jones', 50000)               # E
        pat.giveRaise(.10)                              # P
        print(pat.lastName())                           # D
        print(pat)                                      # P
```

The output of this version tests just its `Manager` class (the imported Example 28-10 tests its own code). As before, a 10% raise is munged to 20% by the custom `giveRaise` here, but other calls are handled by the embedded `Person`, by either direct calls or generic attribute rerouting:

```
$ python3 person-composite.py
Jones
[Person: Pat Jones $60,000]
```

The more important point here is that this `Manager` alternative is representative of a general coding pattern usually known as *delegation*—a composite-based structure that manages a wrapped object and propagates method calls to it.

This pattern works in our example, but it requires about twice as much code and is less well suited than inheritance to the kinds of direct customizations we meant to express. In fact, reasonable Python programmers would almost certainly not code this example this way in practice. `Manager` isn't really a `Person` here, so we need extra code to manually dispatch method calls to the embedded object; operator-overloading methods like `__repr__` must be redefined for reasons explained in the sidebar "Delegating Built-ins—or Not"; and adding new `Manager` behavior is less straightforward since state information is one level removed.

Still, *object embedding*, and design patterns based upon it, can be a good fit when embedded objects require more limited interaction with the container than direct customization implies. A controller layer, or *proxy*, like this alternative `Manager`, for example, might come in handy if we want to adapt a class to an expected interface it does not support, or trace or validate calls to another object's methods (indeed, we will use a nearly identical coding pattern when we study *class decorators* later in the book).

Moreover, a `Department` class prototype like that in Example 28-12 could *aggregate* other objects in order to treat them as a set.

Example 28-12. person-department.py (aggregate embedded objects into a composite)

```
from person_10 import Person, Manager            # Examp

class Department:
    def __init__(self, *args):
        self.members = list(args)                # Manag

    def addMember(self, person):
        self.members.append(person)

    def giveRaises(self, percent):               # Apply
        for person in self.members:
            person.giveRaise(percent)

    def showAll(self):                           # Displ
        for person in self.members:
            print(person)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    pat = Manager('Pat Jones', 50000)

    development = Department(bob, sue)           # Embed
    development.addMember(pat)
    development.giveRaises(.10)                  # Runs
    development.showAll()                        # Runs
```

When run, the department's `showAll` method lists all of its contained objects after updating their state in true polymorphic fashion with `giveRaises`:

```
$ python3 person-department.py
[Person: Bob Smith $0]
[Person: Sue Jones $110,000]
[Person: Pat Jones $60,000]
```

Interestingly, this code uses both inheritance *and* composition—`Department` is a composite that embeds and controls other objects to aggregate, but the embedded `Person` and `Manager` objects themselves use inheritance to customize. As another example, a GUI might similarly use *inheritance* to customize the behavior or appearance of labels and buttons, but also *composition* to build up larger packages of embedded widgets, such as input forms, calculators, and text editors. The class structure to use depends on the objects you are trying to model—in fact, the ability to model real-world entities this way is one of OOP's strengths.

Design issues like composition are explored in Chapter 31, so we'll postpone further investigations for now. But again, in terms of the basic mechanics of OOP in Python, our `Person` and `Manager` classes already tell the entire story. Now that you've mastered the basics of OOP, though, developing general tools for applying it more easily in your scripts is often a natural next step—and the topic of the next section.

DELEGATING BUILT-INS—OR NOT

Notice how the delegation-based `Manager` class of Example 28-11 redefines the `__repr__` run by `print`, instead of allowing it to be caught by `__getattr__` like other attributes. In general, classes cannot intercept and delegate operator-overloading method attributes used by *built-in operations* without redefining them this way. Although we know that `__repr__` is the only such name used in our specific example, this is a broader issue for delegation-based classes.

Recall that built-in operations like printing and addition implicitly invoke operator-overloading methods such as `__repr__` and `__add__`. Built-in operations like these, however, do *not* route their implicit attribute fetches through generic attribute managers: neither `__getattr__` (run for undefined attributes, and used here) nor `__getattribute__` (run for all attributes, and covered later) is invoked. Moreover, the implicit `object` superclass at the top of all class trees defines defaults that can preclude `__getattr__` too.

Hence, `Manager` must redefine `__repr__` redundantly, in order to route printing to the embedded `Person` object. Comment out this class's `__repr__` method to see this live—the `Manager` instance then prints with a default instead of our custom display, because `__getattr__` is never run for printing:

```
$ python3 person-composite.py
Jones
<__main__.Manager object at 0x10a292c30>
```

Technically, built-in operations begin their implicit search for method names at the *class*, skipping the instance entirely. By contrast, explicit by-name attribute fetches are always routed to the *instance* first. Classes also inherit a default `__repr__` from their automatic `object` superclass that would prevent `__getattr__` from running too, but the more absolute `__getattribute__` doesn't intercept the name either, and other methods *not* in `object` also won't be caught by `__getattr__`, like `__add__` for `+`.

This was a mod in Python 3.X, but isn't a showstopper: delegation-based classes can still redefine operator-overloading methods to delegate them to

wrapped objects, either manually or via tools or superclasses. It's also too advanced to explore further in this tutorial, but watch for it to crop up in Chapter 38, be solved in Chapter 39, and make a cameo appearance as a special case in the formal inheritance definition of Chapter 40.

Step 6: Using Introspection Tools

Let's make one final tweak before we throw our objects onto a database. As they are, our classes are complete and demonstrate most of the basics of OOP in Python. They still have two remaining issues we probably should iron out, though, before we go live with them:

- First, if you look at the display of the objects as they are right now, you'll notice that when you print `pat` the `Manager`, the display labels it as a `Person`. That's not technically incorrect, since `Manager` is a kind of customized and specialized `Person`. Still, it would be more accurate to display an object with the most specific (that is, *lowest*) class possible: the one an object is made from.
- Second, and perhaps more importantly, the current display format shows *only* the attributes we include in our `__repr__`, and that might not account for future goals. For example, we can't yet verify that `pat`'s job name has been set to `mgr` correctly by `Manager`'s constructor, because the `__repr__` we coded for `Person` does not print this field. Worse, if we ever expand or otherwise change the set of attributes assigned to our objects in `__init__`, we'll have to remember to also update `__repr__` for new names to be displayed, or it will become out of sync over time.

The last point means that, yet again, we've made potential extra work for ourselves in the future by introducing *redundancy* in our code. Because any disparity in `__repr__` will be reflected in the program's output, this redundancy may be more obvious than the other forms we addressed earlier; still, avoiding extra work in the future is a generally good thing.

Special Class Attributes

We can address both issues with Python's *introspection tools*—special attributes and functions that give us access to some of the internals of objects' implementations. These tools are somewhat advanced and generally used more by people writing tools for other programmers to use than by

programmers developing applications. Even so, a basic knowledge of some of these tools is useful because they allow us to write code that processes classes in generic ways. In our code, for example, there are two hooks that can help us out, both of which were introduced near the end of the preceding chapter and used in earlier examples:

- The built-in *`instance.__class__`* attribute provides a link from an instance to the class from which it was created. Classes in turn have a `__name__`, just like modules, and a `__bases__` sequence that provides access to superclasses. We can use these here to print the name of the class from which an instance is made rather than one we've hardcoded.
- The built-in *`object.__dict__`* attribute provides a dictionary with one key/value pair for every attribute attached to a namespace object (including modules, classes, and instances). Because it is a dictionary, we can fetch its keys list, index by key, iterate over its keys, and so on, to process all attributes generically. We can use this here to print every attribute in any instance, not just those we hardcode in custom displays, much as we did in Chapter 25's module tools.

We met the first of these categories in the prior chapter, but here's a quick review at Python's interactive prompt with the latest versions of our *person.py* classes (Example 28-10). Notice how we load `Person` at the interactive prompt with a `from` statement here—class names live in and are imported from modules, exactly like function names and other variables:

```
>>> from person import Person
>>> pat = Person('Pat Jones')
>>> pat                                          # Run pc
[Person: Pat Jones $0]

>>> pat.__class__                                # Show p
<class 'person.Person'>
>>> pat.__class__.__name__
'Person'

>>> list(pat.__dict__.keys())                    # Attrib
['name', 'job', 'pay']

>>> for key in pat.__dict__:
        print(key, '=>', pat.__dict__[key])      # Index

name => Pat Jones
```

```
job => None
pay => 0

>>> for key in pat.__dict__:                    # obj.at
        print(key, '=>', getattr(pat, key))     # Runs c

…same as prior output…
```

As noted briefly in the prior chapter, some attributes accessible from an instance might not be stored in the `__dict__` dictionary if the instance's class defines a `__slots__` : an optional and relatively obscure feature of classes that stores attributes sequentially in the instance; may preclude an instance `__dict__` altogether; and which we won't study in full until Chapter 32. Since slots really belong to classes instead of instances, and since they are rarely used in any event, we can reasonably ignore them here and focus on the normal `__dict__` .

As we do, though, keep in mind that some programs may need to catch exceptions for a missing `__dict__` , or use built-in functions like `hasattr` , `getattr` , and `dir` if its users might deploy slots. As you'll learn in Chapter 32, the next section's code won't fail if used by a class with slots (its lack of them is enough to guarantee a `__dict__`) but slots—and other "virtual" attributes—won't be reported as instance data.

A Generic Display Tool

We can put these interfaces to work in a superclass that displays accurate class names and formats all attributes of an instance of any class. This superclass is coded in Example 28-13—it's a new, independent module named *classtools.py*. Because its `__repr__` display overload uses generic introspection tools, it will work on *any instance*, regardless of the instance's attributes set. And because this is a class, it automatically becomes a general formatting tool: thanks to inheritance, it can be mixed into *any class* that wishes to use its display format.

As an added bonus, if we ever want to *change* how instances are displayed we need only change this class—every class that inherits its `__repr__` will automatically pick up the new format when next used by a program.

Example 28-13. classtools.py (new)

```
"Assorted class utilities and tools"

class AttrDisplay:
    """
    Provides an inheritable display overload method tha
    instances with their class names and a name=value p
    each attribute stored on the instance itself (but r
    inherited from its classes). Can be mixed into any
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append(f'{key}={getattr(self, key)}')
        return ', '.join(attrs)

    def __repr__(self):
        return f'[{self.__class__.__name__}: {self.gath


if __name__ == '__main__':

    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2

    class SubTest(TopTest):
        pass

    X, Y = TopTest(), SubTest()      # Make two instanc
    print(X)                         # Show all instanc
    print(Y)                         # Show lowest clas
```

Notice the docstrings here—because this is a general-purpose tool, we want to add some functional documentation for potential users to read. As we saw in Chapter 15, docstrings can be placed at the top of simple functions and modules, and also at the start of classes and any of their methods; the `help`

function and the PyDoc tool extract and display these automatically. We'll revisit docstrings for classes in Chapter 29.

When run directly, this module's self-test makes two instances and prints them; the `__repr__` defined here shows the instance's class, and all its attributes' names and values, in sorted attribute name order:

```
$ python3 classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]
```

Another design note here: because this class uses `__repr__` instead of `__str__` its displays are used in all contexts, but its clients also won't have the option of providing an alternative low-level display—they can still add a `__str__`, but this applies to `print` and `str` only. In a more general tool, using `__str__` instead limits a display's scope, but leaves clients the option of adding a `__repr__` for a secondary display at interactive prompts and nested appearances. We'll follow this alternative policy when we code expanded versions of this class in Chapter 31; for this demo, we'll stick with the all-inclusive `__repr__`.

Instance Versus Class Attributes

If you study the `classtools` module's self-test code long enough, you'll notice that its class displays only *instance attributes*, attached to the `self` object at the bottom of the inheritance tree; that's what `self`'s `__dict__` contains. As an intended consequence, we don't see attributes inherited by the instance from classes above it in the tree.

For example, the attribute `TopTest.count` used as an instance counter in this file's self-test code is a *class attribute* that lives only on the class: it's assigned in the class block like methods, and referenced in methods by explicit class name—much like explicit calls to customized class methods. It's also inherited by instances, but inherited class attributes are attached to the class only, not copied down to instances. There's more on such nonmethod class attributes in the next chapter; the main point here is that `count` won't be displayed by `AttrDisplay`.

If you ever do wish to include inherited attributes too, you can climb the `__class__` link to the instance's class, use the `__dict__` there to fetch class attributes, and then iterate through the class's `__bases__` attribute to climb to even higher superclasses, repeating as necessary. If you're a fan of simple code, running a built-in `dir` call on the instance instead of using `__dict__` and climbing would have much the same effect, since `dir` results include inherited names in sorted results lists (along with built-in `__X__` methods that are easy to filter out as usual):

```
$ python3
>>> from person_10 import Person
>>> pat = Person('Pat Jones')

>>> list(pat.__dict__)
['name', 'job', 'pay']

>>> [a for a in dir(pat) if not a.startswith('__')]
['giveRaise', 'job', 'lastName', 'name', 'pay']
```

In the interest of space, we'll leave optional display of inherited class attributes with either tree climbs or `dir` as suggested experiments for now. For more hints on this front, though, watch for the *classtree.py* inheritance-tree climber we will write in Chapter 29, and the *lister.py* attribute listers and climbers we'll code in Chapter 31.

Name Considerations in Tool Classes

One last subtlety here: because our `AttrDisplay` class in the `classtools` module is a general tool designed to be mixed into other arbitrary classes, we have to be aware of the potential for unintended *name collisions* with client classes. As is, the code assumes that client subclasses may want to use both its `__repr__` and `gatherAttrs`, but the latter of these may be more than a subclass expects—if a subclass innocently defines a `gatherAttrs` name of its own, it will likely break our class, because the lower version in the subclass will be used instead of ours.

To see this for yourself, add a `gatherAttrs` to `TopTest` in the file's self-test code; unless the new method is identical, or intentionally customizes the original, our tool class will no longer work as planned—

`self.gatherAttrs` within `AttrDisplay` searches anew from the `TopTest` instance:

```
class TopTest(AttrDisplay):
    …
    def gatherAttrs(self):          # Replaces methc
        return 'Oops'
```

This isn't necessarily bad—sometimes we want other methods to be available to subclasses, either for direct calls or for customization this way. If we really meant to provide a `__repr__` only, though, this is less than ideal.

To minimize the chances of name collisions like this, Python programmers often prefix methods not meant for external use with a *single underscore*: `_gatherAttrs` in our case. This isn't foolproof (what if another class defines its own "private" `_gatherAttrs`, too?), but it's usually sufficient, and it's a common Python naming convention for methods internal to a class.

A better solution would be to use *two underscores* at the front of the method name only: `__gatherAttrs` for us. Python automatically expands such names to include the enclosing class's name, which makes them truly unique when looked up by the inheritance search. This is a feature usually called *pseudoprivate class attributes*, which we'll expand on in Chapter 31 and deploy in an expanded version of this class there. For now, we'll make both our methods available.

Our Classes' Final Form

Now, to use the preceding section's generic display tool in our classes, all we need to do is import it from its module, mix it in by inheritance in our top-level class, and get rid of the more specific `__repr__` we coded before. The new display overload method will be inherited by instances of `Person`, as well as `Manager`; `Manager` gets `__repr__` from `Person`, which now obtains it from the `AttrDisplay` coded in another module. Example 28-14 codes the final version of our *person.py* file with these changes applied, and imports and uses the separate Example 28-13.

Example 28-14. person_14.py (final)

```
"""
Record and process information about people.
Run this file directly to test its classes.
"""
from classtools import AttrDisplay                      #

class Person(AttrDisplay):                              #
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

    def lastName(self):                                 #
        return self.name.split()[-1]

    def giveRaise(self, percent):                       #
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)         #

    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    pat = Manager('Pat Jones', 50000)
    pat.giveRaise(.10)
```

```
print(pat.lastName())
print(pat)
```

As this is the final revision, we've added a few *comments* here to document our work—docstrings for functional descriptions and `#` for smaller notes, per best-practice conventions, as well as *blank lines* between methods for readability—a generally good style choice when classes or methods grow large, which has been resisted earlier for these small classes, in part to save space and keep the code more compact.

When we run this code now, we see all the attributes of our objects, not just the ones we hardcoded in the original `__repr__`. And our final issue is resolved: because `AttrDisplay` takes class names off the `self` instance directly, each object is shown with the name of its closest (lowest) class—`pat` displays as a `Manager` now, not a `Person`, and we can finally verify that job name is correctly filled in by the `Manager` constructor:

```
$ python3 person_14.py
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Pat Jones, pay=60000]
```

This is the more useful display we were after. From a larger perspective, though, our attribute display class has become a *general tool*, which we can mix into any class by inheritance to leverage the display format it defines. Further, all its clients will automatically pick up future changes in our tool. Later in the book, we'll explore even more powerful class tool concepts, such as decorators and metaclasses; along with Python's many introspection tools, they allow us to write code that augments and manages classes in structured and maintainable ways.

Step 7 (Final): Storing Objects in a Database

At this point, our work is almost complete. We now have a *two-module system* that not only implements our original design goals for representing people but also provides a general attribute display tool we can use in other programs in the future. By coding functions and classes in module files, we've ensured that they naturally support reuse. And by coding our software as classes, we've ensured that it naturally supports extension.

Although our classes work as planned, though, the objects they create are not real database records. That is, if we kill Python or our program, our instances will disappear—they're transient objects in memory and are not stored in a more permanent medium like a file, so they won't be available in future program runs. It turns out that it's easy to make instance objects more permanent, with a Python feature called *object persistence*—making objects live on after the program that creates them exits. As a final step in this tutorial, let's make our objects permanent.

Pickles and Shelves

Object persistence is implemented by three standard-library modules, installed with Python itself:

`pickle`
: Serializes arbitrary Python objects to and from a string of bytes

`dbm`
: Implements an access-by-key filesystem for storing strings

`shelve`
: Uses the other two modules to store Python objects on a file by key

We used `pickle` and noted `shelve` briefly in Chapter 9 when we studied file basics. They provide simple yet useful data-storage options. Although we can't do them complete justice in this tutorial or book, they are straightforward enough that a brief introduction should suffice to get you started.

The pickle module

The `pickle` module is a general formatting and deformatting tool: given a nearly arbitrary Python *object* in memory, it converts the object to a string of bytes, which it can use later to reconstruct the original object in memory. The `pickle` module can handle most Python objects—including lists, dictionaries, nested combinations thereof, and class instances. The latter are especially useful things to pickle because they provide both data (attributes) and behavior (methods); the combination is roughly equivalent to "records" and "programs."

Because `pickle` is so general, it can replace extra code you might otherwise write to create and parse custom text-file representations for your objects. By storing an object's pickle string on a file, you effectively make it persistent: simply load and unpickle it later to re-create the original object.

The shelve module

Although it's easy to use `pickle` by itself to store objects in simple flat files and load them from there later, the `shelve` module provides an extra layer of structure that allows you to store pickled objects by *key*. `shelve` translates an object to its pickled string with `pickle` and stores that string under a key in a `dbm` file; when later loading, `shelve` fetches the pickled string by key and re-creates the original object in memory with `pickle`. To your script a shelf of pickled objects looks like a *dictionary*—you index by key to fetch, assign to keys to store, and use dictionary tools such as `len`, `in`, and `dict.keys` to get information. Shelves automatically map most dictionary operations to pickled objects stored in a file.[2]

In fact, to your script, the main coding difference between shelves and normal dictionaries is that you must *open* shelves initially and must *close* them after making changes. The net effect is that `shelve` provides a simple database for storing and fetching native Python objects by keys, and thus makes them persistent across program runs. It does not support query tools such as SQL (but Python's `sqlite3` standard-library module does) and does not have some advanced features found in enterprise-level databases such as true transaction processing (but other Python database tools do). It does, however, store native Python objects that may be processed with the full power of the Python language once they are fetched back by key.

Storing Objects on a shelve Database

Pickles and shelves are somewhat advanced topics, and we won't go into all their details here; you can read more about them in the standard-library manuals. This is all simpler in code than narrative, though, so let's jump right in.

First up is a new script to store objects of our classes on a shelf. Since this is a new file, we'll need to import our classes in order to make instances to store. We used `from` to load a class earlier, but as noted in the prior chapter, there are two ways to get a class from a file (class names are variables like any other, and not at all magic in this context). As an abstract refresher:

```
import person                                   # Loac
bob = person.Person(…)                          # Go t

from person import Person                       # Loac
bob = Person(…)                                 # Use
```

We'll use `from` to load in our script, just because it's less to type. To also keep this simple, we'll duplicate some of the self-test lines from *person.py* that make instances of our classes, so we have something to store (we won't fret over the test-code redundancy in this simple demo). Once we have some instances, it's almost trivial to store them on a shelf. We simply import the `shelve` module, open a new shelf with an external filename, assign the objects to keys in the shelf, and close the shelf when we're done because we've made changes—all per Example 28-15.

Example 28-15. makedb.py (store Person objects in a shelve database)

```
from person_14 import Person, Manager           # Loac
bob = Person('Bob Smith')                       # Re-c
sue = Person('Sue Jones', job='dev', pay=100000)
pat = Manager('Pat Jones', 50000)

import shelve
db = shelve.open('persondb')                    # File
for obj in (bob, sue, pat):                     # Use
```

```
    db[obj.name] = obj                          # Stor
db.close()                                      # Clos
```

Notice how we assign objects to the shelf using their own names as keys. This is just for convenience; in a shelf, the *key* can be any string, including one we might create to be unique using tools such as process IDs and timestamps (available in the `os` and `time` standard-library modules). The only rule is that the keys must be strings and should be unique since we can store just one object per key—though that object can be a list, dictionary, or other object containing many objects itself.

In fact, the *values* we store under keys can be Python objects of almost any sort: built-in types like strings, lists, tuples, and dictionaries, as well as user-defined class instances, and nested combinations of all of these and more. For example, the `name` and `job` attributes of our objects could be nested dictionaries and lists as in earlier incarnations in this book (though this would require a bit of redesign to the current code).

That's all there is to it—if this script has no output when run, it means it probably worked; we're not printing anything, just creating and storing objects in a file-based database:

```
$ python3 makedb.py
```

Exploring Shelves Interactively

At this point, there are one or more files in the current directory whose names all start with *persondb*. The actual files created can vary, and just as in the built-in `open` function, the filename in `shelve.open()` is relative to the current working directory (CWD) unless it includes a directory path. Wherever they are stored, these files implement a keyed-access file that contains the pickled representation of our three Python objects. Don't delete these files—they are your database and are what you'll need to copy or transfer when you back up or move your storage.

You can inspect the shelf's files either from a file explorer, console shell, or Python REPL, but they are binary hash files, and most of their content makes little sense outside the context of the `shelve` module. For example, Python's standard-library `glob` module allows us to get directory listings to verify the

shelf (it's just one file on macOS, *persondb.db*), and we can open it in binary mode to explore stored bytes:

```
>>> import glob
>>> glob.glob('persondb*')
['persondb.db']
>>> print(open('persondb.db', 'rb').read())
b'\x00\x06\x15a\x00\x00\x00\x02\x00\x00\x04\xd2\x00\x00
```

This content can vary, but it's nearly impossible to decipher here, and doesn't exactly qualify as a user-friendly database interface. To verify better, we can write another script, or poke around our shelf at the interactive prompt. Because shelves are Python objects containing Python objects, we can process them with normal Python syntax and development modes. Here, the REPL effectively becomes a *database client*:

```
$ python3
>>> import shelve
>>> db = shelve.open('persondb')                    # Reop
>>> len(db)                                         # Thre
3
>>> list(db.keys())                                 # keys
['Bob Smith', 'Sue Jones', 'Pat Jones']

>>> pat = db['Pat Jones']                           # Fetc
>>> pat.lastName()                                  # Runs
'Jones'
>>> pat                                             # Runs
[Manager: job=mgr, name=Pat Jones, pay=50000]

>>> for key in sorted(db):                          # Iter
        print(key, '=>', db[key])

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Pat Jones => [Manager: job=mgr, name=Pat Jones, pay=500
Sue Jones => [Person: job=dev, name=Sue Jones, pay=1000
```

Notice that we don't have to import our `Person` or `Manager` classes here in order to load or use our stored objects. For example, we can call `pat`'s `lastName` method freely, and get its custom print display format automatically, even though we don't have the `Person` class in scope here.

This works because when Python pickles a class instance, it records the instance's `self` attributes, along with the names of the class it was created from and the module where that class lives. When an instance is later fetched from the shelf and unpickled, Python automatically reimports the class by name and makes a new instance of it having the previously stored instance attributes.

The upshot of this scheme is that class instances automatically acquire all their class behavior when they are loaded in the future. We have to import our classes only to make *new* instances, not to process existing ones. Although a deliberate feature, this scheme has somewhat mixed consequences:

- The *downside* is that classes and their module's files must be *importable* when an instance is later loaded. That is, picklable classes must be coded at the top level of a module file that is accessible from a directory listed on the `sys.path` module search path (and shouldn't live in the topmost script files' module `__main__` unless they're always in that module when used). Because of this external file requirement, some programs pickle simpler objects such as dictionaries or lists, especially if they are to be transferred across networks.
- The *upside* is that changes in a class's source code file are automatically picked up when instances of the class are loaded again. There is often no need to update stored objects themselves since updating their class's code changes their behavior.

Shelves have other well-known limitations covered in Python's library manual. For simple object storage, though, shelves and pickles are easy-to-use tools for personal databases, program configurations, and more.

Updating Objects on a Shelf

One last script: let's write a program that updates an instance (record) each time it runs, to show that our objects really are *persistent*—that their current values are available every time a Python program runs. The file coded in Example 28-16 prints the database and gives a raise to one of our stored objects on each run. If you trace through what's going on here, you'll notice that we're getting a lot of utility "for free"—printing our objects automatically employs the general `__repr__` overloading method, and we give raises by calling the `giveRaise` method we wrote earlier. This all just works for objects based on OOP's inheritance model, even when they live in a file.

Example 28-16. updatedb.py (modify Person object in a shelve database)

```
import shelve
db = shelve.open('persondb')                # Reopen she

for key in sorted(db):                      # Iterate to
    print(key, '\t=>', db[key])             # Prints wit

sue = db['Sue Jones']                       # Index by k
sue.giveRaise(.10)                          # Update in
db['Sue Jones'] = sue                       # Assign to
db.close()                                  # Close afte
```

Because this script prints the database when it starts up, we have to run it at least twice to see our objects change. Here it is in action, displaying all records and increasing `sue`'s pay each time it is run (it's a pretty good script for `sue`; something to schedule to run regularly as a `cron` job perhaps?):

```
$ python3 updatedb.py
Bob Smith     => [Person: job=None, name=Bob Smith, pay
Pat Jones     => [Manager: job=mgr, name=Pat Jones, pay
Sue Jones     => [Person: job=dev, name=Sue Jones, pay=

$ python3 updatedb.py
Bob Smith     => [Person: job=None, name=Bob Smith, pay
Pat Jones     => [Manager: job=mgr, name=Pat Jones, pay
Sue Jones     => [Person: job=dev, name=Sue Jones, pay=

$ python3 updatedb.py
Bob Smith     => [Person: job=None, name=Bob Smith, pay
Pat Jones     => [Manager: job=mgr, name=Pat Jones, pay
Sue Jones     => [Person: job=dev, name=Sue Jones, pay=
```

Again, what we see here is a product of the `shelve` and `pickle` tools we get from Python, and of the behavior we coded in our classes ourselves. And once again, we can verify our script's work at the interactive prompt—the shelf's equivalent of a database client:

```
$ python3
>>> import shelve
>>> db = shelve.open('persondb')              # Reopen c
```

```
>>> rec = db['Sue Jones']                    # Fetch ob
>>> rec
[Person: job=dev, name=Sue Jones, pay=133100]
>>> rec.lastName(), rec.pay
('Jones', 133100)
```

For another example of object persistence in this book, watch for the sidebar "Why You Will Care: Classes and Persistence". It stores a somewhat larger composite object in a flat file with `pickle` instead of `shelve`, but the effect is similar. For more details and examples of pickles, see also Chapter 9 (file basics) and Chapter 37 (string tools and Unicode), as well as Python's manuals.

Future Directions

And that's a wrap for this chapter's tutorial. At this point, you've seen all the basics of Python's OOP machinery in action, and you've learned ways to avoid redundancy and its associated maintenance issues in your code. You've built full-featured classes that do real work. As an added bonus, you've made them real database records by storing them in a Python shelf, so their information lives on persistently.

There is much more we could explore here, of course. For example, we could extend our classes to make them more realistic, add new kinds of behavior to them, and so on. Giving a raise, for instance, should in practice verify that pay increase rates are between zero and one—an extension we'll add when we meet decorators later in this book. You might also mutate this example into a personal contacts database, by changing the state information stored on objects, as well as the classes' methods used to process it. We'll leave this a suggested exercise open to your imagination.

We could also expand our scope to use tools that either come with Python or are freely available in the open source world. For instance, GUIs, websites, or apps would make our database more accessible to nonprogrammers, and other database interfaces like `sqlite3` and content-representation schemes like JSON offer additional possibilities.

While this chapter has hopefully sparked your interest for future exploration, such topics are of course beyond the scope of this tutorial and this book at large. If you want to explore any of them on your own, see the web, Python's

standard-library manuals, and follow-up application texts. First, though, let's return to class fundamentals and finish up the rest of the Python core-language story.

Chapter Summary

In this chapter, we explored all the fundamentals of Python classes and OOP in action, by building upon a simple but real example, step by step. We added constructors, methods, operator overloading, customization with subclasses, and introspection-based tools, and we met concepts such as composition, delegation, and polymorphism along the way.

In the end, we took objects created by our classes and made them persistent by storing them on a `shelve` object database—a simple system for saving and retrieving native Python objects by key. While exploring class basics, we also encountered multiple ways to factor our code to reduce redundancy and minimize future maintenance costs.

In the next chapters of this part of the book, we'll resume our study of the details behind Python's class model and investigate its application to some of the design concepts used to combine classes in larger programs. Before we move ahead, though, let's work through this chapter's quiz to review what we covered here. Since we've already done a lot of hands-on work in this chapter, we'll close with a set of mostly theory-oriented questions designed to make you trace through some of the code and ponder some of the bigger ideas behind it.

Test Your Knowledge: Quiz

1. When we fetch a `Manager` object from the shelf and print it, where does the display format logic come from?
2. When we fetch a `Person` object from a shelf without importing its module, how does the object know that it has a `giveRaise` method that we can call?
3. Why is it so important to move processing into methods, instead of hardcoding it outside the class?
4. Why is it better to customize by subclassing rather than copying the original and modifying?

5. Why is it better to call back to a superclass method to run default actions, instead of copying and modifying its code in a subclass?
6. Why is it better to use tools like `__dict__` that allow objects to be processed generically than to write more custom code for each type of class?
7. In general terms, when might you choose to use object embedding and composition instead of inheritance?
8. What would you have to change if the objects coded in this chapter used a dictionary for names and a list for jobs, as in similar examples earlier in this book?
9. How might you modify the classes in this chapter to implement a personal contacts database in Python?

Test Your Knowledge: Answers

1. In the final version of our classes, `Manager` ultimately inherits its `__repr__` printing method from `AttrDisplay` in the separate `classtools` module, and two levels up in the class tree. `Manager` doesn't have one itself, so the inheritance search climbs to its `Person` superclass; because there is no `__repr__` there either, the search climbs higher and finds it in `AttrDisplay`. The class names listed in parentheses in a `class` statement's header line provide the links to higher superclasses.
2. Shelves (really, the `pickle` module they use) automatically relink an instance to the class it was created from when that instance is later loaded back into memory. Python reimports the class from its module internally and creates a new instance with the previously stored attributes. This way, loaded instances automatically obtain all their original methods (like `lastName`, `giveRaise`, and `__repr__`), even if we have not imported the instance's class into the loading scope.
3. It's important to move processing into methods so that there is only one copy to change in the future, and so that the methods can be run on any instance. This is Python's notion of *encapsulation*—wrapping up logic behind interfaces, to better support future code maintenance. If you don't do so, you create code redundancy that can multiply your work effort as the code evolves in the future.
4. Customizing with subclasses reduces development effort. In OOP, we code by *customizing* what has already been done, rather than copying or

changing existing code. This is the real “big idea” in OOP—because we can easily extend our prior work by coding new subclasses, we can leverage what we’ve already done. This is much better than either starting from scratch each time, or introducing multiple redundant copies of code that may all have to be updated in the future.

5. Copying and modifying code *doubles* your potential work effort in the future, regardless of the context. If a subclass method needs to perform default actions coded in a superclass method, it’s much better to call back to the original through the superclass’s name (or the `super` built-in) than to copy its code. This also holds true for superclass constructors. Again, copying code creates redundancy, which is a major issue as code evolves.
6. Generic tools can avoid hardcoded solutions that must be kept in sync with the rest of the class as it evolves over time. A generic `__repr__` print method, for example, need not be updated each time a new attribute is added to instances in an `__init__` constructor. In addition, a generic `print` method inherited by all classes appears and need be modified in only one place—changes in the generic version are picked up by all classes that inherit from the generic class. Again, eliminating code *redundancy* cuts future development effort; that’s one of the primary assets classes bring to the table.
7. Inheritance is best at coding extensions based on direct customization (like our `Manager` specialization of `Person`). Composition is well suited to scenarios where multiple objects are aggregated into a whole and directed by a controller-layer class. Inheritance passes calls *up* to reuse, and composition passes *down* to delegate. Inheritance and composition are not mutually exclusive; often, the objects embedded in a controller are themselves customizations based upon inheritance.
8. Not much since this was really a first-cut prototype, but the `lastName` method would need to be updated for the new name format; the `Person` constructor would have to change the job default to an empty list; and the `Manager` class would probably need to pass along a job list in its constructor instead of a single string (self-test code would change as well, of course). The good news is that these changes would need to be made in just one place—in our classes, where such details are encapsulated. Apart from their object-construction calls, the database scripts should work as is, as shelves support arbitrarily nested data.
9. The classes in this chapter could be used as boilerplate “template” code to implement a variety of types of databases. Essentially, you can repurpose them by modifying the constructors to record different attributes and

providing whatever methods are appropriate for the target application. For instance, you might use attributes such as `name`, `address`, `birthday`, `phone`, `email`, and so on for a contacts database, and methods appropriate for this purpose. A method named `sendmail`, for example, might use Python's standard-library `smtplib` module to send an email to one of the contacts automatically when run (see Python's manuals or application-level books for more details on such tools). The `AttrDisplay` tool we wrote here could be used verbatim to print your objects because it is intentionally generic. Most of the `shelve` database code here can be used to store your objects, too, with minor changes.

[1] And no offense to any managers in the audience, of course. This joke was once delivered at a Python class in New Jersey, and nobody laughed. The organizers later revealed that the attendees were all managers evaluating Python. Hence the silence.

[2] Terminology note: this book now uses the noun "shelf" for the object storage managed by module `shelve`, and its plural "shelves" for more than one "shelf" managed by `shelve`. In the past, the module's verb name "shelve" was also confusingly used as the noun—much to the chagrin of this book's editors over the years, both electronic and human.