Chapter 23

Objects, Classes and Inheritance

“Mathematicians do not study objects, but relations between objects.” — Henri Poincare

“Blessed are the young for they shall inherit the national debt.” — Herbert Hoover

Object-oriented programming is one of those buzzwords which is difficult to nail down precisely. Reading an explanation of this in, say Wikipedia, quickly opens up more buzzwords and abstractions until it seems like going around in circles. Why all the fuss? To understand what objects (and classes) mean for a programming language, it helps to look briefly at the history of how languages treat data with higher-level operations than manipulating bits and bytes. However, even after learning of this history, there is still the question of why it is important to learn about objects. The simple answer is that, for better or worse, most modern software and their libraries of modules now depend on using objects. So, to use much of the useful software out there (and there is a vast amount of great software), one needs to understand objects and associated concepts.

A bit of history. Back in Chapter 4 there is justification for Python types beyond simple numbers and bits. Python tuples and lists can have any kind of item, including strings, dictionaries, and lists. Other languages don’t have so much freedom. Typically the other programming languages have arrays, or something like lists, which only permit one kind of item (a list only of numbers, or only of strings). There is one practical concern for any language, which is the problem of representing data given by the application, be it business, entertainment, or scientific purpose. Often the data is naturally seen as either a table or a list of records. In a table, there are rows, which resemble records. A spreadsheet is a natural example. Records are common in business and government, where records correspond to individuals (taxpayers), companies, or
Objects and Classes

manufactured appliances (inventory). A feature of a record is that it has fields. Usually, each field has a name (somewhat like a column heading in a table), and an application “type” such as a date, a monetary value, a quantity, checkmarks (paid/unpaid), etc. Using Python, it’s easy to represent a record as using the list type, but in designing applications one might also need to think about how a record would be stored on some permanent media. A low-level language such as assembler or C enables programmers to precisely lay out a record in terms of the bytes of memory and the placement of fields in memory. The C language uses a concept of a data structure, called a struct in C. Similarly, arrays have a precise layout in memory in C.

The simple view of data as records only goes so far to help us represent application information. In a genealogy application, family trees have to be represented; some family trees are large, some are small, so putting a family tree in a single record does not seem practical. If each member of the family tree has a record, then there needs to be some way to relate the records (father, mother, sibling) and group the records (different family names). If records are deleted or arbitrarily modified, the family tree might not make sense. In an effort to avoid making a mess of everything just using records, some theorists proposed abstract data types (ADTs) that not only structure data, but define all the operations that are allowed to read, write, delete and create items of the ADT.

Beyond even abstract data types comes the realization that there are common patterns of applications and computing systems. Two business applications might be quite similar up to a certain level of detail, so it would be sensible to use a common base for their software. Graphical applications often use ideas based around geometry and hardware ideas (pixels, shading). Even if two graphical applications are quite different, certain aspects of them will be alike enough to warrant using the same basic ADTs and algorithms, though customized for efficiency and particular quirks. This is the fundamental realization behind object-oriented programming: the syntax of a programming language can help automate the reuse and customization of ADTs. One principal idea is that of a class hierarchy, wherein all the ADTs in the universe of programming are categorized into various branches, viewed as an evolutionary tree of data structures and operations on them. Different languages have their own hierarchies (Java has its standard library, Microsoft has defined a hierarchy in the .NET framework).

Jargon

This section is quite abstract and may be hard to understand in a single reading. It may help to scan the terminology, go on to the next section, and then come back to this section after seeing a few examples.
Python has syntax allowing a program (module or script) to define a class, which is similar in spirit to a new data type. You can define as many classes (i.e., new data types) as needed. Once a class has been defined, a program can create instances of that class. The situation is similar to the list type in Python. A program can create as many lists as need, and each will have list as its type. Another term for an instance of a class is object: every object is an instance of a class.

Objects are essentially packages of data. In most object-oriented languages the data within an object can only be viewed or changed using methods defined for the object. The definitions of the methods are found in the class from which the object comes (remember, an object is an instance of a class). Python has a less strict interpretation of objects and classes. Objects belong to classes, however objects are completely mutable: the data should be viewed and manipulated using methods, however ordinary assignment statements can read and write data of objects as well.

Inside an object, the data items are typically ordinary basic things such as numbers, characters, strings, or other simple data. Each of the items in the object is a variable named in the class definition for that object. Most object-oriented programming languages call the variables and methods of a class members of that class; in Python these are often called attributes rather than members.

The collected experience of writing software with classes shows that some classes strongly resemble others: the methods are similar, and perhaps the data of objects in the respective classes looks about the same. To take advantage of this observation, programming languages enable a new class to be defined in terms of an existing one. Roughly speaking, the idea is to say “define class X to be like class Y, with a few exceptions.” This notion of defining one class to be like another is called inheritance. Classes can thereby have a parent-child relation. Class X could have class Y as a parent class. The conventional name for a parent class is superclass, whereas the conventional name for a child class is subclass. In most object-oriented programming languages, a parent can have many children, but a child can only have one parent; Python actually does allow a child class to have multiple parent classes (that goes well beyond what this chapter covers). The practical consequence of a parent-child relation is that, by default, class X will have all the attributes (variables and methods) of class Y. There are ways for the definition of class X to override what are Y’s members, if that’s what’s needed in customizing X.

**Elementary Objects in Python**

The simplest example of creating an object, letting g refer to the object, and showing its Python type is:

```python
>>> g = object()
```

Copyright © 2010 Ted R. Herman, all rights reserved.

249
Objects and Classes

>>> type(g)
<type 'object'>

Unfortunately, there’s not much that can be done with such a simple object: it is an instance of class that has no attributes. To make a more meaningful object, we first need to define a class. The simplest definition of a class, here given the name “point” is:

class point():
    pass

The pass statement does nothing; it is only put there because Python expects to have at least one statement within any class definition (otherwise there would be a syntax error). Here’s an interactive example of creating an instance of the class point:

>>> g = point()
>>> type(g)
<type 'instance'>
>>> g.x = 50
>>> g.y = 200
>>> h = g
>>> h is g
True
>>> h.x
50
>>> h.x + g.y
250

The first line creates a point object and lets variable g refer to this new object. The assignment g.x = 50 creates an attribute x of the object. The subsequent lines that assign to h and refer to attributes show that objects are treated like other mutables (lists and dictionaries), as described in Chapter 15. Above, h is an alias of g; they both refer to the same object.

Method Definition

Methods are functions defined within a class. This section shows different methods for class point; suppose p is an instance of point. To call a method, a program would use this syntax:

r = p.mean()

This assigns the result of invoking method mean for object p. Within class point there needs to be a definition of the mean method. For instance,
class point():
def mean(self):
    return (self.x + self.y)/2.0

The definition of mean is unusual: there is a parameter self, but the expression p.mean() has no argument. The explanation is that self represents the object in question, p in this case. Though the variable name p precedes the method mean() in the expression p.mean(), Python treats this as if mean(p) were the expression matching up arguments to the method definition. Another example shows a method requiring an argument:

class point():
def mean(self):
    return (self.x + self.y)/2.0
def right(self,amount):
    self.x += amount
    return

Using this version of point, the expression p.right() would be an error, because self would be p, but there is nothing to match up with amount in the definition. Proper would be p.right(20), which will let parameter amount bind with the number 20. You might also notice that methods can change the values of the variables within an object: p.right(20) increases p.x by 20.

Initializing Method

One method name with special significance is the "__init__" method. Nearly all class definitions are written to have such a method.

class point():
def __init__(self,a,b):
    self.x, self.y = a, b
def right(self,amount):
    self.x += amount

Here is an interactive example using the class point.

```python
>>> p = point(30,80)
>>> p.x
30
>>> p.y
80
>>> p.right(20)
>>> p.x
50
```
In the first line, `point(30,80)` creates an instance of `point`, letting `p` be a reference to this new object. The arguments 30, 80 match up to parameters of the `__init__` method, so that the `x` and `y` variables of the new object have the desired initial values. General terminology for an initializing method is object constructor, since the purpose is to construct a new instance of the class and set up various attributes for later use.

**Containment**

Object attributes can be variables, and these variables could in fact be references to other objects. As a warmup to illustrating this potential of objects, here's a small example showing that a list can have objects as items.

```python
given p = point(0,100) 
given Triangle = [ point(-50,0), point(50,0), p ] 
given point1 = Triangle[0] 
given point1.x: -50 
given point1 is Triangle[0]: True 
given Triangle[2].y: 100
```

The assignment to `Triangle` above shows that creation of an object can be within expressions, which put the object inside a list. A reference to an object behaves like a reference to the other mutable types in Python, dictionaries and lists.

**Line Objects**  In geometry, a line is defined by two (distinct) points. A Python class to make a line could be:

```python
class Line():
    def __init__(self,a,b):
        self.point1, self.point2 = a,b
        if a.x==b.x and a.y==b.y:
            raise ValueError
    def length(self):
        dx = self.point2.x-self.point1.x
        dy = self.point2.y-self.point1.y
        return (dx*dx + dy*dy)**0.5
```

The constructor method `__init__` for `Line` needs two arguments, both of them point objects. Since a line cannot be defined with a single `(x,y)` point, the
constructor raises an error if both arguments refer to the same \((x, y)\) point. The \texttt{length} method calculates the distance between the two points defining the line. Other methods like \texttt{slope} and \texttt{intercept} could easily be defined as well. Using the \texttt{Line} class is straightforward:

```python
>>> L = Line(point(1,1),point(8,-5))
>>> L.point2.y
-5
>>> L.length()
9.2195445729
```

The way that Python evaluates \texttt{L.point2.y} is left-to-right: the \texttt{point2} attribute of object \texttt{L} is itself a \texttt{point} object, which has a \texttt{y} attribute. An equivalent way to write this would be \((\texttt{L.point2}).y\).

## Subclass

This chapter’s exposition of classes and objects is limited; the example here just shows some elementary syntax for class inheritance.

```python
class ColoredLine(Line):
    def setColor(self,newcolor):
        self.color = newcolor
    def getColor(self):
        try:
            return self.color
        except:
            return None
```

The example defines a class \texttt{ColoredLine} that inherits the attributes of its parent class, named in the definition: the \texttt{Line} class. The \texttt{ColoredLine} class possesses the \texttt{Line} attributes plus new attributes, \texttt{color}, \texttt{setColor} and \texttt{getColor}.

```python
>>> L = Line(point(0,0),point(5,1))
>>> L.getColor()
AttributeError: Line instance has no attribute 'getcolor'
```

```python
>>> M = ColoredLine(point(0,0),point(5,1))
>>> print(M.getColor())
None
>>> M.setColor(27)
>>> M.getColor()
27
```

Only a \texttt{ColoredLine} instance has methods \texttt{setColor} and \texttt{getColor}.
Going Further

The syntax for inheritance (defining a new class in terms of an existing parent class) is more complex than the simple example above might suggest. Some difficulties arise from having constructor methods in both parent and child classes; we do not explain this any further — the only reason to show it is that you may come across this syntax while looking at Python programs you encounter. Built-in functions super(), isinstance(), and issubclass() assist in the definition of hierarchical classes. For further information about these functions and the general topic of class hierarchies in Python, a more complete and advanced text should be studied. The only need for hierarchy in this chapter is to exploit some features in the Python library of modules: some of them require that you define a new class in terms of a class defined in a module. Chapter 22 has an example of this: the MyHandler class is defined as a subclass of the BaseHTTPRequestHandler class, so that it can customize behavior of a standard HTTP server.

Period Syntax

Consider all the uses of the period (dot, “.”) in Python:

- The decimal point is part of float notation: 2.5, 1.305e-7, etc.
- Method syntax, which is a kind of function invocation, uses the dot: ''.join(X), Evar.index("ing"), M.sort(), etc.
- Names of variables, classes, and functions in modules imported using the import statement require a period to qualify the name. (And in Python3, a period can be used to refer to directory and subdirectory relation where a module is located.) Examples of this are math.pi, math.sqrt(), random.choice().
- Attributes of objects use the p.x notation.

When periods occur in expressions, recall that the default rule for Python syntax is left-to-right evaluation. As Python evaluates such an expression, the type of the term under evaluation determines what is expected next. If Python determines that R is a module name in evaluating R.t, then t had better be some variable, function, or class within module R. Similarly, if x is an object reference in the expression x.a, then a must be an attribute of x. Technically, this holds for an expression like ''.join(V) because the empty string ‘’ is an instance of the “string” class, and the method join is defined in the string class, so ''.join is an attribute:

```python
>>> type(''.join)
<type 'builtin_function_or_method'>
```
In expression \(M.e.b\), it could be that \(M\) is a module, \(e\) is an object reference inside of the module, and \(b\) is an attribute of \(e\). Even more complicated expressions are allowed by Python, so long as they follow the syntax rules and make sense in terms of the types and the rules for using dot listed above.

\[(Tref[i][j]).(Ftab("S",k)).summary(V.y[b:b+8])\]

In parsing this rather messy expression, Python could first determine that \(Tref[i][j]\) refers to a module, then determine that \(Ftab("S",k)\) returns an object reference within that module, and that \(summary\) is a method of the object — so the expression would make sense. Of course this complicated way of expressing an idea is not recommended.

**Date/Time Objects**

The Python `datetime` module is based on objects. As with all Python modules, the only reliable documentation is that found at [www.python.org](http://www.python.org); it is somewhat cryptic documentation, however with knowledge of objects and Python syntax, and perhaps some examples found with a bit of searching, these modules are very useful. The following script is an exercise using `datetime` objects.

```python
import datetime
import date

DayBirth = datetime.date(1992,8,4)  # date of birth
WeekDays = "Mon Tue Wed Thu Fri Sat Sun".split()
day = WeekDays[ DayBirth.weekday() ]
S = "born on a {0}".format(day)
print(S)

Now = datetime.date.today()  # current date/time
Age = Now - DayBirth  # creates timedelta object
S = "current age in days since birth is {0}".format(Age.days)
S = S.format(Age.days)
print(S)
```

One interesting feature shown above is the calculation of age, \(Age = Now - DayBirth\). The `datetime` module actually extends Python’s minus operator (“-”) so that objects can be subtracted (in this case, two `date` objects). The result creates a `timedelta` object. This hints at how Python can define classes and instances that essentially behave as new data types for the language.

**Regular Expressions**

*Some people, when confronted with a problem, think "I know, I’ll use regular expressions." Now they have two problems.*

— Jamie Zawinski
The concept of *regular expressions* is fundamental to much of the day-to-day work in Information Technology and the software industry. The essential idea is easy to grasp: consoles, or command shells, typically allow one to see all the files that begin with “T” by entering

```
> ls T*
T.pl Tcpserv.py Token.bak Tankfill.tgz Transactor TAX.doc
```

The “*” is a *wildcard* character, meaning that it can stand for any string of characters. Hence, filenames like *Temp.txt* and *Tomorrow.data* would be listed, whereas files that don’t begin with T would not be shown. Many search engines allow similar syntax to limit and tailor search results.

Regular expressions are supported by libraries or modules in most programming languages. Some programming languages even incorporate regular expressions into the syntax of the language itself, notably Perl, where using regular expressions is a way of life, so to speak. Regular expressions go well beyond “*”, typically using many special characters (*, [ ], [ ], *, ( ), ?, and others) to control how text to be searched will match up with some pattern expression. On the one hand, regular expressions enable very concise, flexible and powerful matching of patterns to text data, which is quite useful for tasks of searching, data extraction, conversion, and general preparation of information.

On the other hand, the notation and conventions of regular expressions are a mini-language that is easy to forget, can differ slightly from one programming language to the next, and has a very cryptic appearance. Whatever one might conclude about regular expression usage, it’s an important programming idiom in current software systems and worth knowing about.

The official Python documentation includes a “Regular Expression HOWTO”, which should be consulted for those planning to use this feature in Python. What follows is just one example of the power of regular expressions. The task in the example is to read a file *dorian.txt* and find all strings in the file consisting of lowercase characters, starting with “a” and ending with “ly”, making a list of these.

```python
import re
T = open("dorian.txt").read()
regobject = re.compile("a[a-z]*ly")
matches = regobject.findall(T)
print(matches)
```

The pattern for searching through the file text above is the string "a[a-z]*ly", which stands for anything that begins with a, following by any number of characters in the lowercase alphabet (a–z), followed by ly. The output of the example is too lengthy to reproduce here (the file *dorian.txt* contains over 75,000 words). The first few lines of the output are:
Network programs may fetch webpages, which use HTML, a language to mark up text for display and browser interaction. While it’s possible to write Python functions that scan HTML and extract meaning from the data, it’s usually easier to use a Python module to do this. The following example is a script that processes HTML in a string (alternatively it could be in a file object, as shown in Chapter 22). The script uses the Python2 version of HTML processing, but Python3’s version is nearly the same.

```python
import HTMLParser

class LinkParser(HTMLParser.HTMLParser):
    def handle_starttag(self, tag, attrs):
        if tag == 'a':
            linktuple = attrs[0]
            href, url = linktuple
            self.mylinks.append(url)

T = '''<html><body bgcolor="gray">
<h1>My Title</h1><p>Welcome to Webpage. The <a href="http://www.python.org">Python Link</a> is a good resource. Sometimes the <a href="http://en.wikipedia.org">Wikipedia page</a> is helpful.</p></body></html>'''

# create a parser object
p = LinkParser()
p.mylinks = [] # collect links
p.feed(T)
for url in p.mylinks:
    print(url)
```

Observe that to use the HTMLParser, a script must define a class that inherits from the HTMLParser class, customizing a method that is called repeatedly during a scan of some HTML input. The input is “fed” to a parser object,
which is created first (above, it’s variable p). Thanks to Python’s flexibility, the script adds a new attribute mylinks, a list to collect all the URLs in a webpage. Object-oriented software purists would not write the script this way: a purist would instead create the mylinks attribute inside the __init__ method, however doing so would entail using super(), which goes beyond what this chapter covers. The output of the script above is

http://www.python.org
http://en.wikipedia.org

Terminology Review

Object-oriented programming is rich with jargon. Motivations start with records, fields within records, structured data and abstract data types. The notion of classes and instances of classes, commonly called objects, incorporates inheritance so that classes can form a hierarchy, in which a class may have parent and child classes (superclass and subclass). Within a class, there are members, which could be data or methods; Python notation uses a period (dot) after an object reference for an attribute; an attribute may be a variable or may be a method. When one class inherits from another, all the members automatically carry over from the parent, unless the definition of the child overrides them. Most standard Python modules use class/object techniques; some modules require that applications using them define subclasses which override methods to customize their behavior.

Exercises

(1) Find and read documentation of Python’s webbrowser module. Then try the following, interactively:

```python
>>> import webbrowser
>>> T = webbrowser.get()
>>> T.open("http://www.python.org")
```

This should cause a web browser to start and display a web page. Above, the variable T is a reference to a browser “controller” object. Write a script that launches a web browser and opens several tabs with different pages.

(2) This exercise is for Unix/Linux systems only (might work on a Mac OS, too). The module for the exercise is subprocess, which enables a Python script to issue typical commands like ls (list directory), tail (get the last lines of a file), mv (move or rename a file or directory), and hundreds more. The commands can even be new ones that you invent. The subprocess
not only lets you issue the commands within Python, but any number
of parameters to these commands can be given as a list of strings, and
the response from the command will be captured as a string, where the
Python script can extract information. This kind of facility to call up
system commands, get responses, and process data, approaches the real
meaning of script – it can automate what people might have to otherwise
do manually, but which is a regular enough activity to merit automation.
Here is an example using subprocess:

```python
import sys
import subprocess

# create a Popen object, which runs the unix command
# ls -l /opt
# listing what's in the /opt directory.
P = subprocess.Popen(["ls","-l","/opt"],stdout=subprocess.PIPE)
# run the Popen object and wait until it's done: the
# communicate() method returns a tuple (A,B) where both
# A and B are strings, with A being the command's normal
# output and B having error messages, if any
out,error = P.communicate()
# go through lines in out, and nicely format them
for line in out.split(
    sys.stdout.write("\t" + line + 
sys.stdout.write("-----------------------") # make marker at end
```

Modify this example so that it changes options on the ls command (you
can read about the options by reading a manual page on ls, searching
online). Another experiment could be to extract just some information
from the string in out and summarize that.