4 Using Objects

In this chapter we explore objects and code re-use. Python is an object-oriented language and learning to use objects can make programming fun and productive. In this chapter we'll explore object-oriented programming by using the turtle module.

If we had to write every program from scratch, we wouldn't be able to get very much done. Part of the fun of programming is using something someone else has written to solve a problem quickly. Another fun aspect of programming is writing code that others may want to use in their programs. In fact, programmers sometimes become famous among their peers by writing code that turns out to be very valuable: people like Yukihiro Matsumoto [2], who created the Ruby programming language, or Robin Milner [6] who described the type inference system used by Standard ML, or Guido van Rossum the creator of the Python Programming Language [10]. There are many, many computer scientists that could be named here.

Python makes it easy for programmers who want to share code with others to do just that. A *module* is a file containing Python code. When a programmer needs to use code another programmer wrote, he or she can import the module containing the code they want to use into their program. Modules can be imported into other modules so one programmer can easily use code that another programmer wrote. One such module is called *turtle*. The *turtle* module includes code that helps us draw figures in the sand. A turtle can walk around a beach dragging his or her tail in the sand or raising that tail. When the tail is down, the turtle leaves a track. When the tail is up the turtle leaves no trail. With this simple analogy we can draw some pretty interesting pictures. The idea has been around since at least the late 1960s when Seymour Papert added turtle graphics to the Logo programming language [4]. Gregor Lingl, an Austrian high school teacher, has implemented a version of turtle graphics for Python that now is part of the Python programming environment.

To use a module it needs to be imported into your program. There are two ways to import a module. The decision of which to use is partly based on convenience and partly based on safety of your program. The safe way to import a module is to write `import module` where *module.py* is the name of a module. The module must be in the current directory or in one of the directories where your installation of Python knows to look. When importing a module in this way you must prefix any use of

K.D. Lee, *Python Programming Fundamentals*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6642-9_4

code within the module with the module name. If you want to call a function or use a type, t, that is defined in the imported module, you must write *module.t*. This is safe because there will never be the possibility of using the same name within two different modules since all names must be *qualified* with the module name. Using *qualified* names makes importing safe, but is not the most convenient when writing code.

Example 4.1 Here is a program that imports the turtle code and uses it to draw a square.

```python
import turtle

t = turtle.Turtle()
screen = t.getscreen()
t.forward(25)
t.left(90)
t.forward(25)
t.left(90)
t.forward(25)
t.left(90)
t.forward(25)
screen.exitonclick()
```

If you are going to try this code, DO NOT call it turtle.py. If you name your own program the same as a module name, then Python will no longer import the correct module. If you already did this you must delete the turtle.pyc file in your folder and rename your module to something other than turtle.py.

Example 4.1 imports the turtle module using *import turtle*. Once the module is imported, a Turtle object can be created. In this case, the programmer must write *turtle.Turtle*() to create an object of type *Turtle*. Because the Turtle type or *class* resides in the turtle module the fully qualified name of *turtle.Turtle*() must be written to create a Turtle object. Figure 4.1 shows the *turtle* reference pointing to a *Turtle* object just like integer variables are references that point to *int* objects and string variables are references that point to *str* objects. Initializing a Turtle object and making a reference point to it is just like creating any other object in Python.

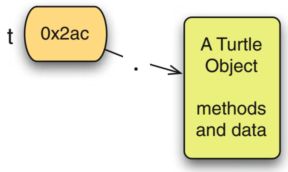

Fig. 4.1 A turtle object

Practice 4.1 Write some code that uses a for loop to draw a square using the turtle module.

A more convenient way to import a module is to write *from module import* *. In this case we could import the turtle module by writing *from turtle import* *. This imports the turtle module as before but merges all the names of functions, types, and classes in the turtle module with the names of functions, variables, and types in your program.

Example 4.2 Here is a program that draws a pentagon using the other form of import.

```python
from turtle import *

t = Turtle()
screen = t.getscreen()
t.forward(25)
t.left(72.5)
t.forward(25)
t.left(72.5)
t.forward(25)
t.left(72.5)
t.forward(25)
t.left(72.5)
t.forward(25)
screen.exitonclick()
```

Example 4.2 imports by merging the namespace of the turtle module and the program in the example. Both Examples 4.1 and 4.2 demonstrate how to call a *method* on an *object*. This means that any variables defined in the turtle module will be overridden if they are also defined in the code in Example 4.2. For example, we would want to be careful and not name something *Turtle* in our code since that would mean that we would no longer be able to create a Turtle object in our program. Redefining a name like this is not permanent though. The problem only exists within the program. Once the program terminates, the next time we import the turtle module, the Turtle class would be available again.

Not every class must be imported from a module. Python already makes the *int*, *float*, *bool*, and *str* classes available without importing anything. These classes are called *built-in* classes in Python. But, the *Turtle* class is not built-in. It must be imported from the *turtle* module.

In both examples the variable *t* is a reference that points to a *Turtle* object. The turtle object can be told to do things. Turtles understand certain messages or methods. We've already learned how to call methods on objects in Chap. 3. For instance, we've called the *split* method on a *string* object. Sending a message to a Turtle object is no different. For instance in Example 4.2 we sent the *forward* message to the turtle *t* passing 25 as the number of steps to move forward. The *forward* method, and other methods that turtles understand, are described in Chap. 13. Methods for the *TurtleScreen* class are described in Chap. 14.

Practice 4.2 Write a short program that prompts the user to enter the number of sides of a regular polygon. Then draw a regular polygon with that many sides. You can use the *textinput* method described in Chap. 14 to get the input or you can just use *input* to get the input from the Debug I/O tab of Wing IDE 101.

While actual turtles are slow and perhaps not very interesting, turtle objects can be fun. A turtle object can be used in a lot of different ways. It can change color and width. It can be used to draw filled in shapes. It can draw circles and even display messages on the screen. Turtle graphics is a great way to become familiar with object-oriented programming. The best way to learn about object-oriented programming is just to have fun with it. Refer to Chap. 13 and use it to write some programs that draw some interesting pictures with color, interesting shapes, filled in polygons, etc.

Practice 4.3 Use the turtle module to write a program that draws a 4WD truck like that pictured in Fig. 4.2. A truck consists of two tires and a top of some sort. You should use some color. You may use *penup* and *pendown* while drawing. However, don't use *goto* once you have started drawing. The reason for this will become evident in the exercises at the end of the chapter.

You may want to change color, fill in shapes, etc. Be creative and try things out. Just be sure the last line of your program is *screen.exitonclick*(). Without the call to *screen.exitonclick*() the turtle graphics window may appear to freeze up.

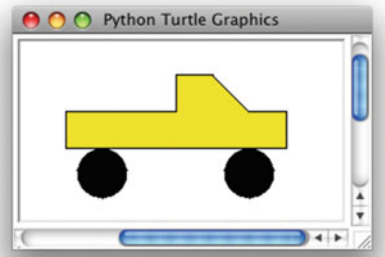

Fig. 4.2 A 4WD truck

4.1 Constructors

To create an object of a certain type or *class* we must write

```
<objectref> = <Class>(<args>)
```

This creates an object of type *Class* and then points the *objectref* variable at the object that was just created. Figure 4.1 shows what happens in memory as a result of executing the *t=Turtle*() line of code in Examples 4.1 and 4.2. Several things happen when we create an object. Python first reserves enough space in memory to hold the object's data. Then, the object is initialized with the data that must be stored in it. All objects have some data associated with them. For instance, a Turtle object knows its current location on the screen, its direction, and its color, among other things. When a Turtle object is created, all the information is stored in the object. This is called *constructing an object* and it happens when we call the constructor. So, when we write the following line of code or similar lines of code for other types of objects:

```
t = Turtle()
```

we are instructing Python to create a Turtle object using the constructor and we make the variable *t* point to the turtle object that was just created. There are lots of constructors that are available to us for creating different types of objects in Python.

Example 4.3 Here are some examples of objects being created using constructors. The types (i.e. classes) *str*, *int*, *float*, *Turtle*, and *list* each have their own constructors. In fact, sometimes a class has more than one constructor. Look at the float examples below. There are at least two ways to create a float object. You can either pass the constructor a string and it will convert the float in the string to a float object, or you can pass an integer to the float constructor.

```python
s = str(6)
x = int("6")
y = float("6.5")
z = float(6)
t = Turtle()
lst = list("a b c")
u = 6
r = "hi there"
```

Except in a few special circumstances, a constructor is always called by writing the name of the class then a left paren, then any arguments to pass to the constructor, followed by a right paren. Calling a *constructor* returns an instance of the *class*, called an *object*. For a few of the built-in classes there is some *syntactic sugar* available for creating objects. In Example 4.3, the variables *u* and *r* are initialized to point to an integer object and a string object, respectively. *Syntactic sugar* makes constructing objects for some of the built-in classes more convenient and it is necessary in some cases. Without some *syntactic sugar*, how would you create an object containing the integer 6?

Practice 4.4 Using Wing, or some other IDE, run the code in Example 4.2. Set a breakpoint at the line where *screen* is initialized. Then, look at the *Stack Data* and specifically at the *t* variable. Expand it out so you can see the state of the turtle and specifically the *_position* of the turtle. This is the (x,y) location of the turtle on the screen. When the turtle is at the peak of the pentagon from Example 4.2 what is its (x,y) location?

4.2 Accessor Methods

When we have an object in our program, we may wish to learn something about the state of that object. To ask for information about an object you must call an *accessor* method. Accessor methods return information about the state of an object.

Example 4.4 To learn the heading of the turtle we might call the *heading* method.

```
import turtle

t = turtle.Turtle()

print(t.heading())
```

Calling the heading method on the turtle means writing *t* followed by a dot (i.e. a period) followed by the name of the method, in this case *heading*. The accessor method, *heading*, returns some information about the object, but does not change the object. Accessor methods do not change the object. They only access the state of the object.

Practice 4.5 Is the *forward* method an accessor method? What about the *xcor* method? You might have to consult Chap. 13 to figure this out.

4.3 Mutator Methods

Mutator methods, as the name suggests, change or mutate the state of the object. Sect. 3.5 introduced the mutability of lists. Mutator methods are called the same as accessor methods. Where an accessor method usually gives you information back, a mutator method may require you to provide some information to the object.

Example 4.5 Here are some calls to mutator methods.

```
turtle.right(90)
turtle.begin_fill()
turtle.penup()
```

One misconception about object-oriented programming is that assigning one reference to another creates two separate objects. This is not the case as is demonstrated by the following code. This isn't a problem if the object doesn't change. However, when the object may be mutated it is important to know that the object is changing and this means that it changes for all references that point at the object.

Example 4.6 Here is an example of one turtle with two different references to it. Both *t* and *r* refer the the same turtle.

```
t = Turtle()
t.forward(50)

r = t

r.left(90)
r.forward(100)

t.left(90)
t.forward(50)
```

In Example 4.6 more than one reference points to the same Turtle object as depicted in Fig. 4.3. Writing $r = t$ does not create a second Turtle. It only points both references to the same Turtle object. This is clear from Example 4.6 when one Turtle seems to pick up where the other left off. In fact, they are the same turtle.

Practice 4.6 How would you create a second Turtle object for *r* if that's really what you wanted?

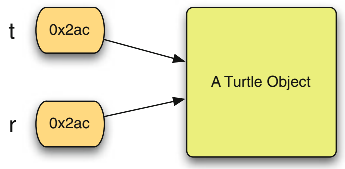

Fig. 4.3 Two references to one object

4.4 Immutable Classes

Section 3.5 first defined immutable classes. An immutable class is a type with no mutator methods. If an object has no mutator methods then it is impossible to tell if two references point to the same object or if they point to different objects. In fact it doesn't really matter since neither reference can be used to change the object. This may happen frequently in Python for objects of type *int*, *float*, *string*, and *bool*. All these classes are immutable. These classes of objects can never be changed once they are created since they have no mutator methods!

Practice 4.7 If strings cannot be changed, what happens in the following code? Draw a picture to show what happens in the following code.

```
x = "hello"
x = x + "world"
print(x)
```

While string objects can't be changed, references can be. That's what happens in the exercise above. *str* objects never change once they are created. Immutable classes are nice to work with because we can forget about their being objects and references and just concentrate on using them without fear of changing the object accidentally.

4.5 Object-Oriented Programming

Turtles are fun to program because they make drawing easy by remembering many of the details of generating computer graphics for us. That's really the motivation behind object-oriented programming and using objects. What we've seen in this short chapter are all the mechanics for creating and using objects. Objects make our lives as programmers simpler. Every object maintains some state information, its data, and every object lets us either access that data through an accessor function or it allows its data to be changed by calling mutator methods. Many objects have both accessor and mutator methods.

The power of object-oriented programming is in the ability to organize the data in our programs into logical entities that somehow make sense. A turtle is a great way to embody many of the elements of graphics programming while giving us a way of visualizing how the turtle works by thinking about how a real turtle might leave marks in the sand.

4.6 Working with XML Files

Now that you know how to use objects and in particular how to use turtle graphics you can put it to use. There are many applications for Turtle graphics. It can be used to create more advanced drawing applications like the one pictured in Fig. 4.4.

The drawing application shown in Fig. 4.4 can save pictures in a file format called XML. XML stands for eXtensible Markup Language. Computer Scientists devised the XML format so data could be stored in a consistent format. Many applications store their data in XML format. Some that you might be familiar with include the Apple iTunes application or the registry in Microsoft Windows. Mac OS X uses it as well in its application structure. XML is popular because the definition of XML makes it possible to add additional elements to an XML file later without affecting code that was written before the new fields were added. This ability to add to an XML file without breaking existing code means there is a huge advantage to using XML as the format for data in practically any application. Being able to write code to extract data from an XML file is a very practical skill.

XML files have a fairly straight-forward structure but also contain a lot of formatting information that is not really part of the data. It would be painful to have to write code that reads an XML file and extracts just the information you need. Fortunately, it is because XML files contain this extra formatting information, often

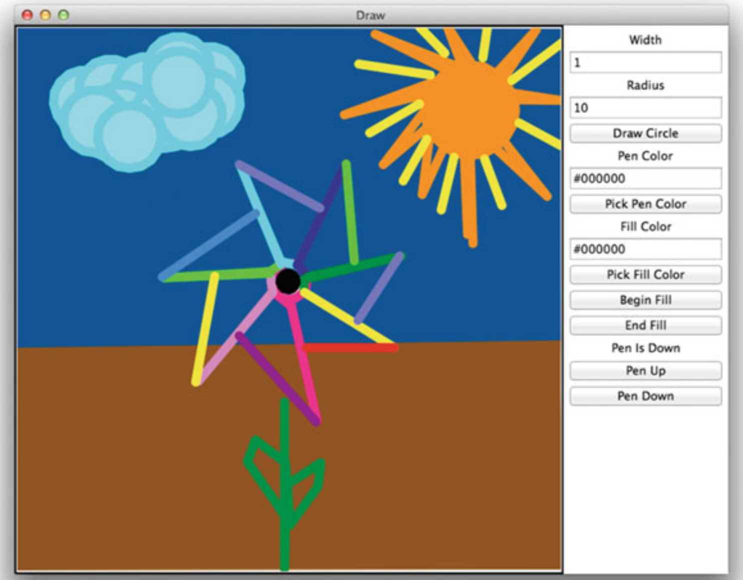

Fig. 4.4 Flower power by Denise M. Lee

called meta-data, that it is possible for someone else to write code that we can use to read an XML file. That code is called an XML parser. Parsing refers to reading data and selecting out the individual components or elements of that data.

To parse an XML file you must import an XML parser. We'll use the minidom XML parser in this text. The import statement looks like this:

```
from xml.dom import minidom
```

Once you have imported the XML parser you create an *Document* object by telling minidom to parse the XML file.

```
xmldoc = minidom.parse("flowerandbg.xml")
```

That's all there is to reading an entire XML file. Looking at Fig. 4.4 it should be clear that the picture is fairly complex. There are many colors and elements to the drawing. Just how is all that data organized?

An XML file starts with a line at the top that helps the parser identify the contents of the file as an XML file. The parser looks for a line that looks something like this.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

The rest of an XML file is composed of one or more elements. And, elements may be nested inside of other elements. Elements almost always consist of two tags with text or other elements nested between the tags. A tag in an XML file is a string of characters that appears within angle brackets (i.e. a less than/greater than sign pair). For instance, this is one element from an XML file with a start-tag and end-tag and the text "PenUp" nested inside the element.

```
<Command>PenUp</Command>
```

Each start-tag has a matching end-tag that ends one element of an XML file. The `<Command>` is the start-tag of this element and the `</Command>` is the end-tag. The matching end-tag always has the same name as the start-tag but is preceeded by a slash. An XML element may also contain attributes. The attributes appear within the XML element's *start-tag* as shown here.

```
<Command x="299.0" y="-45.0" width="1.0"
        color="#804000">GoTo</Command>
```

This element contains the attributes *x*, *y*, *width*, and *color*. Each of these attributes has a value inside the quotes associated with the attribute.

Start-tags and end-tags almost always occur in matching pairs. However, there is one other type of element that consists of just one tag. An element with no nested elements may be written like this.

```
<Command type="PenUp"/>
```

There are no occurrences of empty elements like this in the graphics file in the following example. Most of the time XML elements consist of a start-tag and end-tag pair with possibly nested elements or text between the tags.

Example 4.7 Here is an example of a file saved by a drawing program. This file contains one XML element called *GraphicsCommands*. Within this single XML element are many *Command* elements. These elements represent a subset of the drawing commands used to produce the picture in Fig. 4.4.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<GraphicsCommands>
    <Command color="#804000">BeginFill</Command>
    <Command x="299.0" y="-45.0" width="1.0"
       color="#804000">GoTo</Command>
    <Command x="302.0" y="-297.0" width="1.0"
       color="#804000">GoTo</Command>
    <Command x="-301.0" y="-298.0" width="1.0"
       color="#804000">GoTo</Command>
    <Command x="-300.0" y="-53.0" width="1.0"
       color="#804000">GoTo</Command>
    <Command>EndFill</Command>
    <Command color="#004080">BeginFill</Command>
    <Command x="-300.0" y="300.0" width="1.0"
       color="#004080">GoTo</Command>
    <Command x="299.0" y="299.0" width="1.0"
       color="#004080">GoTo</Command>
    <Command x="300.0" y="-45.0" width="1.0"
       color="#004080">GoTo</Command>
    <Command>EndFill</Command>
    <Command>PenUp</Command>
    <Command x="0.0" y="0.0" width="1.0"
       color="#000000">GoTo</Command>
    <Command>PenDown</Command>
    <Command radius="10.0" width="10"
       color="#ffffff">Circle</Command>
    <Command radius="20.0" width="10"
       color="#ff0080">Circle</Command>
    <Command x="2.0" y="-4.0" width="10.0"
       color="#ff0080">GoTo</Command>
    <Command x="2.0" y="-5.0" width="10.0"
       color="#ff0080">GoTo</Command>
    <Command x="2.0" y="-6.0" width="10.0"
       color="#ff0080">GoTo</Command>
</GraphicsCommands>
```

4.7 Extracting Elements from an XML File

Each element in an XML document has a name. To extract an element you ask for all elements that match a given name. For the drawing application's XML document format we start by getting the *GraphicsCommands* element.

```
graphicsCommands = \
    xmldoc.getElementsByTagName("GraphicsCommands")
```

The code above returns a list of all elements at the top-level of the document that match the tag name *GraphicsCommands*. We know there is only one of these elements in the file, so we can get just the first one by using index 0 into the list.

```
graphicsCommand = \
    xmldoc.getElementsByTagName("GraphicsCommands")[0]
```

The graphicsCommand variable is set to the first, and only, of the matching DOM elements returned by the minidom parser. DOM stands for Document Object Model. Now that we have the graphicsCommand element we can get sub-elements from it. The sub-elements of it are the list of *Command* elements.

```
commands = graphicsCommand.getElementsByTagName("Command")
```

Finally, if we wish to draw the picture stored in the file, we can traverse the *Command* elements with a for loop.

```
for command in commands:
  # Draw the command on the screen.
  # This code is omitted for now.
```

4.8 XML Attributes and Dictionaries

In the XML file presented in Example 4.7 many of the *Command* elements have attributes. For instance, the *BeginFill* command has a *color* attribute. The *GoTo* command on lines 4–5 has attributes *x*, *y*, *width*, and *color*. These attributes provide information about each of their graphics commands. The attribute names of *x*, *y*, *width*, and *color* are called the attribute keys and their values are the strings to which each key is assigned.

To correctly draw the picture in one of these picture XML files, we must be able to access the attributes of a graphics command and use them when drawing the picture. It is possible to access the attributes of an XML element through an *attributes* dictionary.

A dictionary is a little like a list. You can use indexing to look up values within the dictionary just like you use indexing to look up values within a list. The difference is that instead of using only integers as the index values, you can use any value you like. To lookup an attribute in the *attributes* dictionary we use its key.

Example 4.8 A list and a dictionary have similarities. Both data types hold a collection of values. The difference between a list and a dictionary are in the values used to index into them. In a list, the index values must be non-negative integers and the locations within the list are numbered sequentially starting at 0.

Within a dictionary there is no ordering of the index values. An index value, called a *key* when working with dictionaries, can be nearly any value. A dictionary is a list of *key*, *value* pairs. Each key is mapped to a value. Keys must be unique, values do not have to be unique in the dictionary.

Here is some code that creates both a list and a dictionary and demonstrates similar operations on the two datatypes.

```python
lst = [] # an empty list
dct = {} # an empty dictionary

# The append method adds items to a list.
lst.append("Biking")
lst.append("Running")
lst.append("Other")

# The next line adds Sport/Running as a key/value pair
dct["x"] = "299.0"
dct["y"] = "-45.0"
dct["color"] = "#804000"

# We can iterate over a list by using a for loop.
for i in range(len(lst)):
    print(i, lst[i])

# We can iterate over a dictionary using a for loop
# to go through the list of keys to the dictionary.
for key in dct.keys():
    print(key, dct[key])
```

The output when this code is executed is as follows.

```
0 Biking
1 Running
2 Other
y -45.0
x 299.0
color #804000
```

Chapter 12 contains a complete listing of dictionary operators and methods.

4.9 Reading an XML File and Building Parallel Lists

Drawing a picture like the one in Fig. 4.4 is possible if the corresponding XML file is parsed and the graphics commands extracted from it. Imagine that not only do we want to read such a picture, we would like to be able to scale the picture to make it bigger or smaller. It is possible to do this if we store all the graphics commands in lists. We'll store each graphics command and its attributes in separate lists. One list will hold all the graphic command names. Another will hold the *color* attribute of each graphic command. Still another will hold the *x* attribute of each command, and so on.

This technique of using multiple lists to hold data that are related to each other is called *parallel lists*. The lists are in a sense parallel to each other because each list contains information that is related to the others at the same index value within the list. Each index location within the six lists contains the six attributes of one graphics command. Since we will need to go through the data more than once if we are scaling the picture, it makes sense to store this information in parallel lists so we can go through it as often as we need.

If a particular graphic command, like *BeginFill* does not have an attribute, like *x* for instance, then a special value of *None* will be stored at that location in the list. In this way the parallel lists will all have the same length and all the related data for a graphics command will be stored at the same index location within all the lists. We'll name these parallel lists for their attribute names with a *List* attached to the end. So the list of *x* attributes becomes *xList* for example. The graphic command list will just be called *commandList*.

Example 4.9 Code to build these parallel lists is relatively simple. There are five attributes. Each of these attributes corresponds to one list. Line 14 of the code in this example deserves some further explanation. In this line the expression *command.firstChild.data* retrieves the text appearing between the start-tag and end-tag of an XML element. For instance, when examining the element from line three of the XML file in Example 4.7 the *command.firstChild.data* would be "BeginFill". The text in between the tags tells the code in this example which graphics command is represented in each of the XML elements.

```
1  import turtle
2  from xml.dom import minidom
3  xmldoc = minidom.parse("flowerandbg.xml")
4  graphicsCommand = \
5       xmldoc.getElementsByTagName("GraphicsCommands")[0]
6  commands = graphicsCommand.getElementsByTagName("Command")
7  commandList = []
8  xList = []
9  yList = []
10 widthList = []
11 colorList = []
12 radiusList = []
13 for command in commands:
14     commandList.append(command.firstChild.data.strip())
15     attr = command.attributes
16     if "x" in attr:
17         xList.append(attr["x"].value)
18     else:
19         xList.append(None)
20     if "y" in attr:
21         yList.append(attr["y"].value)
22     else:
23         yList.append(None)
24     if "width" in attr:
25         widthList.append(attr["width"].value)
26     else:
27         widthList.append(None)
28     if "color" in attr:
29         colorList.append(attr["color"].value)
30     else:
31         colorList.append(None)
32     if "radius" in attr:
33         radiusList.append(attr["radius"].value)
34     else:
35         radiusList.append(None)
```

The code above is very repetitive on lines 17–36 doing the same thing for each attribute. All that changes is the attribute name and the list to which the value is appended. This code could be rewritten to use two parallel lists of its own, one for the attribute names, and another for the attribute lists. So, we end up with two more lists, the *attrributeList* and the *attributes* list. This shortens the code considerably. This code does exactly the same thing as the code above.

```
import turtle
from xml.dom import minidom
xmldoc = minidom.parse("flowerandbg.xml")
graphicsCommand = \
    xmldoc.getElementsByTagName("GraphicsCommands")[0]
commands = graphicsCommand.getElementsByTagName("Command")
commandList = []
xList = []
yList = []
widthList = []
colorList = []
radiusList = []
attributeList = [xList,yList,widthList,colorList,radiusList]
attributes = ["x","y","width","color","radius"]
for command in commands:
    commandList.append(command.firstChild.data.strip())
    attr = command.attributes

    for i in range(len(attributes)):
        attr = command.attributes
        key = attributes[i]
        if key in attr:
            attributeList[i].append(attr[key].value)
        else:
            attributeList[i].append(None)
```

Notice the way the code above iterates over the attributes. Since the *attributes* list and the *attributeList* list are the same length the *for i in range*(*len*(*attributes*)) generates the index *i* into both parallel lists. When working with parallel lists you always use an indexed *for loop* like this or *while loop* so you have an index that you can use to index into any of the parallel lists.

4.10 Using Parallel Lists to Draw a Picture

Drawing the picture from the XML file means traversing the parallel lists that were built in Sect. 4.9. Each location in the list contains a graphics command like "GoTo" or "Circle". The attributes for a command are stored at the same index in a parallel list. All that's needed is to iterate over these parallel lists and execute the turtle commands that are required to draw the picture.

Example 4.10 To draw the picture in one of the picture XML files it is only necessary to iterate over the parallel lists that were built in Sect. 4.9. The code below uses the *colormode* method. Passing 255 to this method means that colors will be set using hexadecimal numbers. Color hexadecimal number have 6 digits. The first two digits are for the amount of red. The second two digits are the amount of green. The third two digits are the amount of blue. Two hexadecimal digits can range from 00–FF, or 0–255 when converted to decimal values. With 256 different shades of red, green, and blue there are 256^3 different possible colors.

The use of *screen.tracer*(*0*) below means that the picture is drawn instantaneously without any screen updates. This makes the picture just appear when the *screen.update*() method is called. Update must be called to force the update of the screen since the setting *tracer* to 0 means no updates are done automatically.

Notice the use of the *for i in range*(*len*(*commandList*)) below. The parallel lists all have the same length. The only way to traverse all the lists simultaneously is by indexing into the list. So, *i* is used as the index into all the parallel lists.

```
t = turtle.Turtle()
screen = t.getscreen()
screen.colormode(255)
screen.tracer(0)
for i in range(len(commandList)):
    command = commandList[i]
    if command =="PenUp":
        t.penup()
    elif command == "PenDown":
        t.pendown()
    elif command == "GoTo":
        x = float(xList[i])
        y = float(yList[i])
        width = float(widthList[i])
        color = colorList[i]
        t.width(width)
        t.color(color)
        t.goto(x,y)
    elif command == "Circle":
        radius = float(radiusList[i])
        width = float(widthList[i])
        color = colorList[i]
        t.width(width)
        t.pencolor(color)
        t.circle(radius)
    elif command == "BeginFill":
        color = colorList[i]
        t.fillcolor(color)
        t.begin_fill()
    elif command == "EndFill":
        t.end_fill()
    else:
        print("Unknown Command:",command)

screen.update()
screen.exitonclick()
```

4.11 Review Questions

1. What are the two ways to import a module? How do they differ? What are the advantages of each method of importing?
2. How do you construct an object? In general, what do you have to write to call a constructor?
3. What happens when you construct an object?
4. What is the purpose of an accessor method?
5. What is the purpose of a mutator method?
6. Does every class contain both mutator and accessor methods? If so, why? If not, give an example when this is not true.
7. What does an XML file contain?
8. How do you read an XML file in a program?
9. What is an attribute in an XML file? Give an example.
10. What type of value does the method *getElementsByTagName* return when it is called?
11. What is a dictionary?
12. What are parallel lists? Why are they necessary in some cases?

4.12 Exercises

1. Write a program that plots the function
$$g(x) = x^4/4 - x^3/3 - 3x^2$$
You can use the setworldcoordinates method to plot the function on the screen from -20 to 20 on the x-axis and -20 to 20 on the y-axis. When you are done, if you did it right, you should have a screen that looks like Fig. 4.5. To plot the function the x values can go from -20 to 20. The y values can be found by using the definition of the function g. Be sure to include the dots for the units on the graph.
2. Write the program described starting in Sects. 4.9 and 4.10. Create a sample XML file using the draw program found on the text's website. Draw a picture and save it. The file will be in XML format. Save the XML file in the same directory or folder where you save your program. By saving the program and the XML file in the same directory your program will find the XML file when you run it.
3. Write a program as described in the last exercise but after reading the XML file, prompt the user for a scale factor and scale the entire picture by the scale factor. Each (x,y) coordinate must be multiplied by the scale parameter. Draw the picture with its new scale.
4. Write a program as described in last exercise. Get a scale factor from the user. However, this time also prompt the user for a new file name. Then write a new XML file with the new scale factor integrated into it. This can be done one of

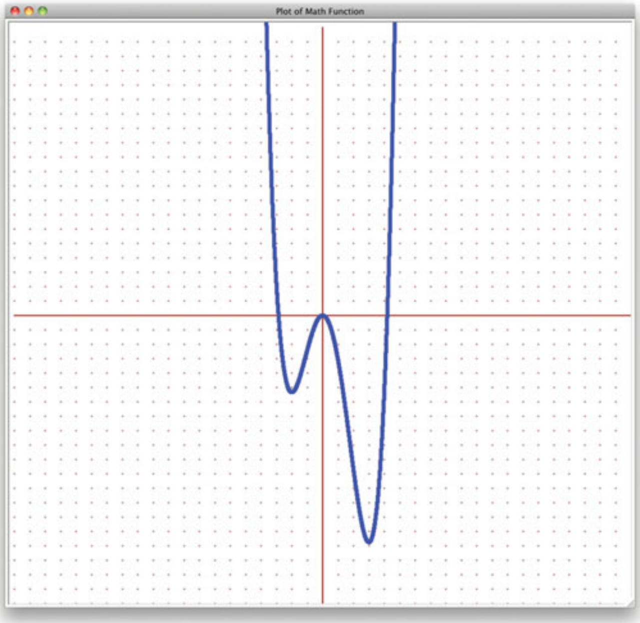

Fig. 4.5 The plot of $g(x) = x^4/4 - x^3/3 - 3x^2$

two ways. You can write the file with all new (*x,y*) coordinates, or you can add a *scale* attribute to the *GraphicsCommands* element.

5. On the website for the text there are three files, *Toyota4Runner.csv*, *Nissan-Versa.csv*, and *SuzukiS40.csv* that all contain gas mileage information for their corresponding vehicles. Write a program that reads this data and plots average miles per gallon in one dimension and time in the other dimension. Since the first line of each file is not a record, but a description of the columns, you might want to use a *while loop* to read the data so you can throw away the first line before starting the while loop.
   HINT: Since each field of the records is in double quotes you can read the line from the file and put square brackets around it as follows.

```
x = "[" + '"Gas","0.0","2010-01-19","10:38 PM", ...' + "]"
y = eval(x)
```

   The call to the *eval* function will force the evaluation of the string. Calling *eval* like this returns a list of the elements; in this case a list of strings as the fields of the record. Using this technique will make parsing the input extremely easy.
   You will want to create datetime objects for each fill up date. You must first import the *datetime* module to create datetime objects. For a discussion of *datetime* objects or you can read about them on the web. Search for "python datetime" to read about the datetime module on the web. You can get the number of days

from two datetime objects by subtracting them and then using the difference as follows.

```
timeDelta = firstDay - lastDay
days = timeDelta.days
```

It might first appear that the difference should be computed as *lastDay - firstDay*. However, this yields a negative number of days so the example in the listing above is correct when computing days.

6. When looking at average EPA MPG for gas powered vehicles there is always a city MPG and a highway MPG, with highway MPG being greater. Since filling the car multiple times within a short amount of time would seem to indicate that a person is taking a trip, there should be a correlation between filling up over short amounts of time (i.e. highway miles) and the observed MPG. Use the *Toyota4Runner.cvs* or the *NissanVersa.cvs* files to plot days since last fill up and observed MPG. You will want to do this as a scatter plot. A scatter plot is simply a dot for each data point. A dot can be made using the *dot* method of the Turtle class. Observe the data that you find there and draw a regression line through that data. A regression line is a best fit line. It minimizes the total distance of points to the line.
   To compute the days since last fill up you will probably want to use the datetime module. See the previous exercise for a discussion of *datetime* objects or you can read about them on the web. Search for "python datetime" to read about the datetime module on the web.
   To draw a regression line you need to keep track of a few things.

   - The sum of all the x values
   - The sum of all the y values
   - The sum of all the x^2 values
   - The sum of all the x*y values

   Decide what your x and y axis represent. Compute the values given above. When you have gathered these values you need to use the values in the formula below.

$$y = \overline{y} + m(x - \overline{x})$$

   where

$$m = \frac{\sum_{i=1}^{n} x_i y_i - n\overline{x}\,\overline{y}}{\sum_{i=1}^{n} x_i^2 - n\overline{x}^2}$$

   All the values you need in the formulas are available in the values you kept track of above. n is the number of data points. $\overline{x}$ is the average of the x values and likewise for $\overline{y}$. The sum of all the x^2 values is the sum of the squares, NOT the square of the sum.
   To plot the regression line you can choose two x values and then compute their corresponding y values given the formula $y = \overline{y} + m(x - \overline{x})$. This will give you

the two end points of the regression line. Once you have the two end points of the line, use the turtle to draw the line between them. Plot this regression line to see the correlation between highway miles and MPG. While this program will compute a linear regression line, it should be noted that the correlation between number of highway miles and MPG is definitely NOT a linear function so observed results should be understood in that context.

7. In practice Problem 4.3 you drew a truck using a turtle. You should not have used any *goto* method calls in that practice problem. In this exercise you are to draw trucks of random size at random places on the screen. To generate random numbers in a program you need to import the *random* module. You create a random number generator as follows:

```
from random import *

rand = Random()
```

Their are three methods that Random objects support that you may want to use:

- **rand.randrange(start, stop, step)**—*start* default is 0, *step* default is 1. It returns a random integer in the range [start, stop) that is on one of the *steps*.
- **rand.randint(start, stop)**—*start* default is 0. It returns a random integer in the range [start, stop).
- **rand.random()**—Returns a random floating point number in the range [0,1).

For this exercise you should repeatedly draw trucks at different locations on the screen. You can use the *goto* method to move to a randomly selected location on the screen. By default the screen goes from −500 to 500 in both directions so generating a screen location in the range −400–400 in both directions will work well.

Once you have moved to a random location on the screen, draw the truck as you did in practice Problem 4.3. However, to make the trucks different sizes, randomly generate a floating point number between 0 and 1 using the *random* method. This random number is a scale for your truck. Multiply each forward or circle argument by the scale when drawing the truck. By multiplying the forward and circle arguments by a number between 0 and 1 you are creating scaled versions of your truck from 0 (no truck at all) to 1 (a full-size truck).

NOTE: Do not multiply turns times the scale. All angles are the same in any scaled version of the truck.

4.13 Solutions to Practice Problems

These are solutions to the practice problems in this chapter. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

4.13.1 Solution to Practice Problem 4.1

```
import turtle

t = turtle.Turtle()
screen = t.getscreen()
for k in range(4):
    t.forward(25)
    t.left(90)

screen.exitonclick()
```

4.13.2 Solution to Practice Problem 4.2

```
from turtle import *

t = Turtle()
screen = t.getscreen()
sides = int(screen.textinput("Polygon", \
          "Please Enter the Number of Sides:"))
for k in range(sides):
    t.forward(200//sides)
    t.left(360/sides)

screen.exitonclick()
```

4.13.3 Solution to Practice Problem 4.3

```
from turtle import *

t = Turtle()
screen = t.getscreen()

t.fillcolor("black")
t.begin_fill()
t.circle(20)
t.end_fill()
t.penup()
t.forward(120)
t.pendown()
t.begin_fill()
t.circle(20)
t.end_fill()
t.penup()
t.left(90)
t.forward(40)
t.right(90)
t.forward(30)
t.right(180)
t.pendown()
t.fillcolor("yellow")
t.begin_fill()
t.forward(180)
t.right(90)
t.forward(30)
t.right(90)
t.forward(90)
t.left(90)
t.forward(30)
```

```
t.right(90)
t.forward(30)
t.right(45)
t.forward(43)
t.left(45)
t.forward(30)
t.right(90)
t.forward(30)
t.end_fill()
t.ht()

screen.exitonclick()
```

4.13.4 Solution to Practice Problem 4.4

The turtle's location is (12.0388, 38.18233) at the peak of the pentagon.

4.13.5 Solution to Practice Problem 4.5

The *forward* method is not an accessor method. The *xcor* method is an accessor method. It accesses the x coordinate of the turtle.

4.13.6 Solution to Practice Problem 4.6

You create a second turtle the same way you created the first.

```
from turtle import *

t = Turtle()
screen = t.getscreen()
t.forward(100)
secondTurtle = Turtle()
secondTurtle.left(90)
secondTurtle.forward(100)
screen.exitonclick()
```

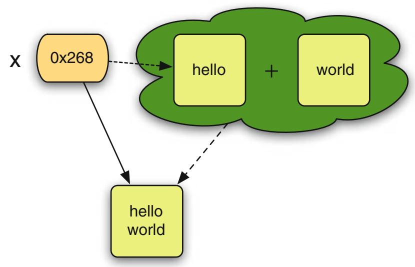

Fig. 4.6 Concatenation of two strings

4.13.7 Solution to Practice Problem 4.7

Figure 4.6 depicts what happens when the following code is executed. This is pretty much identical to what happens with the integers on p. 22 in Chap. 1.

```
x = "hello"
x = x + "world"
print(x)
```