Defining Classes

7

Python is an object-oriented language. This means, not only can we use objects, but we can define our own classes of objects. A class is just another name for a type in Python. We have been working with types (i.e. classes) since the first chapter of the text. Examples of classes are *int*, *str*, *bool*, *float* and *list*. While these classes are all built in to Python so we can solve problems involving these types, sometimes it is nice if we can solve a problem where a different type or class would be helpful.

Classes provide us with a powerful tool for abstraction. Abstraction is when we forget about details of how something works and just concentrate on using it. This idea makes programming possible. There are many abstractions that are used in this text without worrying about exactly how they are implemented. For example, a file is an abstraction. So is a list. In fact, integers are abstractions, too. A turtle is an abstraction that helps us implement Turtle graphics programs. Instead of worrying about how a line gets drawn in a window, we can just move the turtle along the line with its pen down to draw the line. How is this done? It's not important to us when we are using a Turtle. We just know it works.

So, classes are a great tool for programmers because when a programmer uses a class they don't have to worry about the details. But, sometimes we might be able to save time and implement a class that could be useful to us and maybe to someone else as well. When we use a class we don't worry about the details of how an object works. When we implement a class we must first decide what the abstraction is going to look like to the user of it and then we must think about how to provide the right methods to implement the abstraction. When defining or implementing a class, the *user* is either yourself or another programmer that is going to use the class when they create some objects of the class you defined.

Classes provide the definitions for objects. The *int* class defines what integers look like and how they behave in Python. The *Turtle* class defines what a turtle looks like and all the methods that control its behavior. In general, a class defines what objects of its type look like and how they behave. We all know what an integer looks like. Its behavior is the operations we can perform on it. For instance we might want to be able to add two integers together, print an integer, and so on. When we define our own classes we do two things.

K.D. Lee, *Python Programming Fundamentals*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6642-9_7

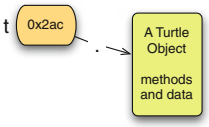

Fig. 7.1 A Turtle object

- A Class defines one or more data items to be included in the objects or instances of the class. These data items are sometimes called the *member data* or *instance variables* of the class. Each instance, or object, will contain the data defined by the class.
- A Class defines the methods that operate on the data items or member data in objects of the class. The methods are functions which are given an object. A method defines a particular behavior for an object.

To understand how objects are created we can look at an example. In Chap. 4 we learned how to create Turtle objects and use them to do write some interesting programs.

Example 7.1 When we execute the code below, Python creates a Turtle object pointed to by the reference *t* as shown in Fig. 7.1.

```
t = Turtle()
```

We have already learned that we could make the turtle go forward 50 units by writing *turtle.forward(50)*. The *forward* function is a method on a Turtle. It is part of the turtle object's behavior. As another example, consider a Circle class. A circle must be drawn on the screen at a particular location. It must be given a radius. It might have a fill color and it might have a width and color for its outline.

7.1 Creating an Object

When an object is created there are two things that must happen: the space or memory of the object must be reserved, and the space must be initialized by storing some values within the object that make sense for a newly created object. Python takes care of reserving the appropriate amount of space for us when we create an object. We must write some code to initialize the space within the object with reasonable values. What are reasonable values? This depends on the program we are writing.

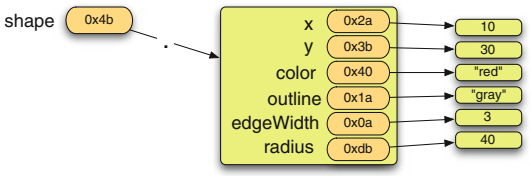

Fig. 7.2 A circle object

Example 7.2 To create a circle we might write something like this.

```
x = 10
y = 30
radius = 40
shape = Circle(x,y,radius,edgeWidth=3, \
            color="red",outline="gray")
```

Creating a circle called *shape* creates an object that contains the data that we give the *constructor* when the circle is created. The constructor is called when we write the class name, followed by the arguments to pass to the constructor. In this case, the call to the constructor is *Circle*(*x, y, radius, width=3, color="red", outline="gray"*). The constructor takes care of putting the given information in the object. Figure 7.2 shows what the data looks like in the object after calling the constructor.

The data in an object doesn't get filled in by magic. We must write some code to do this. When programming in an object-oriented language like Python we can write a *class* definition once and it can be used to create as many objects of that class as we want. To help us do this, Python creates a special reference called *self* that always points to the object we are currently working with. In this way, inside the class, instead of writing the reference *shape* we can write the reference *self*. By using the reference *self* when writing the code for the class, the code will work with any object we create, not just the one that *shape* refers to. We are not stuck just creating one circle object because Python creates the special *self* reference for us. We can create a *shape* and any other circle we care to create by writing just one Circle class.

Example 7.3 The first method of a class definition is called the *constructor* and is named *__init__*. It takes care of filling in the member data inside the object. The *self* reference is the extra reference, provided by Python, that points to the current object. This method gets called in response to creating an object as occurs in the code in Example 7.2.

```
class Circle:
    def __init__(self,x=0,y=0,radius=50,color="transparent", \
                 outline="black",edgeWidth=1):
        self.x = x
        self.y = y
        self.color = color
        self.outline = outline
        self.edgeWidth = edgeWidth
        self.radius = radius
```

In Example 7.3 notice that the formal parameters nearly match the arguments provided when the circle object is created in Example 7.2. The one additional parameter is the extra *self* parameter provided by Python. When the constructor is called, Python makes a new *self* local variable for the *__init__* function call. This *self* variable points at the newly created space for the object. Figure 7.3 shows the run-time stack with the *self* variable pointing at the newly created object. The picture shows what memory looks like just before returning from the *__init__* constructor method. There are two activation records on the run-time stack. The first is the activation record for the function that creates the shape by executing the code in Example 7.2. The second activation record is for the *__init__* function call (i.e. the call to the constructor). When the program returns from the constructor the top activation record will be popped and the *self* reference will go away.

To implement a class we must write the word *class*, the name of the class, and then the methods that will operate on the objects of that class. By convention, the first method is always the constructor. Generally other methods follow and must be indented under the class definition. The class definition ends when the indentation under it ends.

Practice 7.1 Decide what information you would need to implement a Rational class. Rational numbers are numbers that can be expressed as a fraction with an integer numerator and denominator. Then write a class definition for it including a constructor so you can create Rational objects.

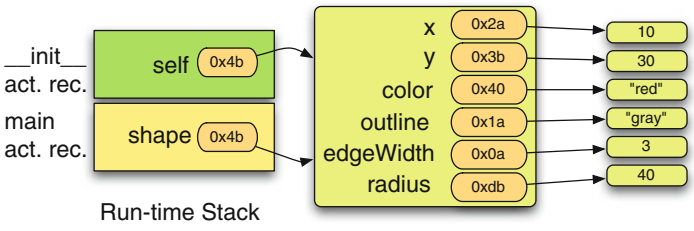

Fig. 7.3 A circle object

Practice 7.2 Assume we want to implement a class for rectangles. A rectangle is created at a particular (x, y) location specifying the lower left corner of the rectangle. A rectangle has a width and height. Write a class definition for the Rectangle class so that a rectangle can be created by writing *box = Rectangle(100, 100, 50, 30)* to create a rectangle at (100, 100) with a width of 50 and a height of 30.

If we have a circle object, it would be nice to draw it on a turtle graphics screen. In addition, we may want to change its color, width, or outline color at some point. These are all actions that we want to perform on a circle object and because they change the object in some way they will become mutator methods when implemented. In addition, we may want to access the x, y, and radius values. These are implemented with accessor methods. The mutator and accessor methods must be defined in the class definition.

Example 7.4 Here is the complete code for the Circle class.

```
class Circle:
    # This is the constructor for the class. It
    # takes the data provided as arguments
    # and stores the data in the object.
    def __init__(self,x=0,y=0,radius=50,color="transparent", \
                 outline="black",edgeWidth=1):
        self.x = x
        self.y = y
        self.color = color
        self.outline = outline
        self.edgeWidth = edgeWidth
        self.radius = radius

    # The draw method is a mutator method, too. It does
    # not store anything in the object, but it uses the turtle
    # and therefore mutates the turtle object.
    def draw(self,turtle):
        turtle.penup()
        turtle.goto(self.x,self.y)
        turtle.width(self.edgeWidth)
        if self.color != "transparent":
            turtle.fillcolor(self.color)
        turtle.color(self.outline)
        turtle.fillcolor(self.color)
        turtle.setheading(0)
        turtle.forward(self.radius)
        if self.color != "transparent":
            turtle.begin_fill()
        turtle.pendown()
        for k in range(500):
            radians = (2*math.pi)*(k/500.0)
            turtle.goto(math.cos(radians)*self.radius+self.x, \
                        math.sin(radians)*self.radius+self.y)
        if self.color != "transparent":
            turtle.end_fill()
        turtle.penup()
```

```
37            turtle.goto(self.x,self.y)
38
39        # The following three methods are mutator methods.
40        # They each take a single value passed to the
41        # method and store it in the object.
42        def setEdgeWidth(self,width):
43            self.edgeWidth = width
44
45        def setFill(self,color):
46            self.color = color
47
48        def setOutline(self,color):
49            self.outline = color
50
51        # The last three methods are accessor methods.
52        # They return three of the fields of the object.
53        def getX(self):
54            return self.x
55
56        def getY(self):
57            return self.y
58
59        def getRadius(self):
60            return self.radius
```

When a method is called on an object the variable is written first, followed by a dot (i.e. period), followed by the method name. So, for instance, to call the *getX* method on the *shape* you would write *shape.getX*(). When you look at the definition of *getX* there is one parameter, the self *parameter*. When you call *getX* it looks like there are no parameters. Python sets *self* to point to the same object that appears on the left side of the dot. So, in this example, the *self* parameter points at the *shape* object because *shape* was written on the left hand side of the dot. The picture in Fig. 7.3 applies to calling the *getX* method as well. When *getX* is called, an activation record is added to the stack with the *self* variable pointing at the object. This is true of all classes in Python. When implementing a class the first parameter to all the methods is always *self* and the object that is on the left hand side of the dot when the method is called is the object that becomes self while executing the method.

Practice 7.3 Complete the Rectangle class by writing a *draw* method that draws the rectangle on the screen. When drawing a rectangle allow the color of the border and the color of the background to be specified. Specify these parameters with default values of black and transparent respectively. Make these parameters keyword parameters with the names *outline* and *color* (for background color).

7.2 Inheritance

A class is an abstraction that helps programmers reuse code. Code reuse is important because it frees us to solve interesting problems while allowing us to forget the details of the classes we use to solve a problem. Code reuse can be achieved between classes as well. When objects are similar in most respects but one is a special case of another the relationship between the classes can be modeled using inheritance. A *subclass* inherits from a *superclass*. When using inheritance, the subclass gets everything that's in the superclass. All data and methods that were a part of the superclass are available in the subclass. The subclass can then add additional data or methods and it can redefine existing methods in the superclass.

Inheritance in Computer Science is like inheritance in genetics. We inherit certain physical characteristics of our birth parents. We may look different from them but typically there are some similarities in hair color, eye color, height and so on. We probably also inherit behaviors from our parents, although this may come from social contact with our parents and isn't necessarily genetic. Inheritance when applied to Computer Science means that we don't have to rewrite all the code of the superclass. We can just use it in the subclass.

Inheritance comes up all over the place in OOP. For instance, the Turtle class inherits from the RawTurtle class. The Turtle class is essentially a RawTurtle except that a Turtle creates a TurtleScreen object if one has not already been created.

Example 7.5 Here is the entire Turtle class.

```
class Turtle(RawTurtle):
    """RawTurtle auto-creating (scrolled) canvas.

    When a Turtle object is created or a function derived
    from some Turtle method is called a TurtleScreen
    object is automatically created.
    """
    _pen = None
    _screen = None

    def __init__(self,
                 shape=_CFG["shape"],
                 undobuffersize=_CFG["undobuffersize"],
                 visible=_CFG["visible"]):
        if Turtle._screen is None:
            Turtle._screen = Screen()
        RawTurtle.__init__(self, Turtle._screen,
                           shape=shape,
                           undobuffersize=undobuffersize,
                           visible=visible)
```

While the code in Example 7.5 is difficult to completely understand out of context, the Turtle class only consists of a constructor, the minimum amount that can be

provided in a derived class. The constructor creates the screen if needed and then calls the RawTurtle's constructor. Every class, whether a derived class or a base class, *must* provide its own constructor. When Python creates an object of a certain class, it needs the constructor to determine how the object is initialized. So, the class Turtle in Example 7.5 truly contains the minimal amount of methods possible for a derived class.

Essentially a Turtle and a RawTurtle are identical. It also turns out that Turtles (and RawTurtles) are based on Tkinter. A TurtleScreen contains a ScolledCanvas widget from Tkinter. To create a RawTurtle object we must provide a ScrolledCanvas for the Turtle to draw on.

Example 7.6 Here is the constructor definition for a RawTurtle.

```
class RawTurtle(TPen, TNavigator):
    """Animation part of the RawTurtle.
    Puts RawTurtle upon a TurtleScreen and provides tools for
    its animation.
    """
    screens = []

    def __init__(self, canvas=None,
                 shape=_CFG["shape"],
                 undobuffersize=_CFG["undobuffersize"],
                 visible=_CFG["visible"]):
```

Because turtle graphics is based on Tkinter, we can write a program that contains widgets including a canvas on which we can draw with turtle graphics! The constructor in Example 7.6 shows us that if we provide a canvas the RawTurtle object will use it. So, we could write a little drawing program that draws circles and rectangles on the screen and integrates other Tk widgets, like buttons for instance.

To begin building a draw application we'll put a ScrolledCanvas on the left side of a window and some buttons to control drawing on the right side. Since we've been looking at a Circle class, we'll start by drawing circles on the screen. It would be nice to provide the radius for the circle. We can do that with an entry field and a StringVar object as was seen in the last chapter.

Example 7.7 Here is some code that creates a ScrolledCanvas widget, a RawTurtle that draws on the canvas, and a Tkinter application that incorporates both. Figure 7.4 shows what the application window looks like when it is run. Notice the use of the class definition for *DrawApp*. Encapsulating all the tkinter application code in a class means that *self* can be used to store variables that need to be globally available to the application. In particular, the *shapeSelection* variable in the object is used and set in multiple places in the class. The main function simply creates a *DrawApp* object and then calls *mainloop* to make the tkinter application start listening for events.

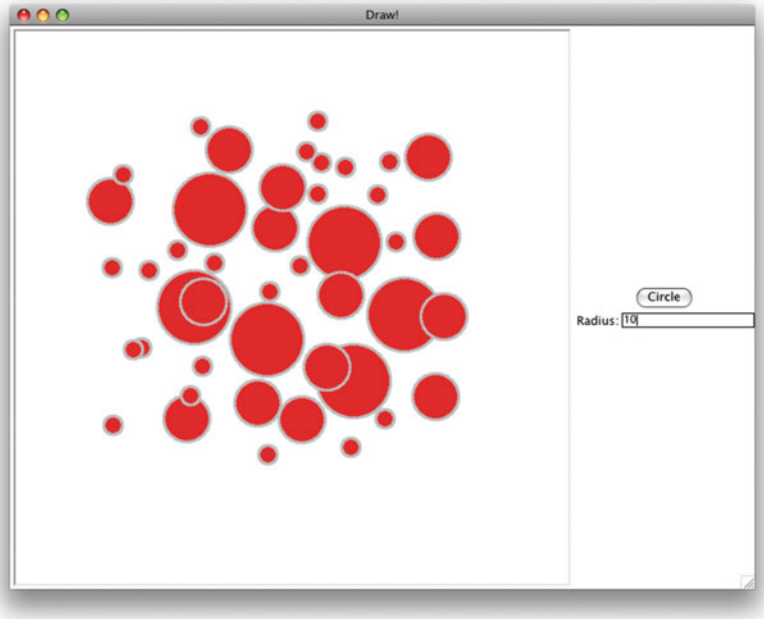

Fig. 7.4 A drawing application

```
from turtle import *
from tkinter import *
import math

noselection = 0
circle = 1

# The Circle class is omitted here.

class DrawApp:
    def __init__(self):
        root = Tk()
        root.title("Draw!")
        self.shapeSelection = noselection
        cv = ScrolledCanvas(root,600,600,600,600)
        cv.pack(side = LEFT)
        aTurtle = RawTurtle(cv)
        screen = aTurtle.getscreen()
        aTurtle.ht()
        screen.tracer(0)

        fram = Frame(root)
        fram.pack(side = RIGHT,fill=BOTH)

        def circCommand():
            print("in circCommand")
            self.shapeSelection = circle

```

```
29          radiusEnt = StringVar()
30          radiusLabel = Label(fram,text="Radius:")
31          radiusLabel.grid(row=2,column=1,sticky=E)
32          radiusEntry = Entry(fram,textvariable=radiusEnt)
33          radiusEntry.grid(row=2,column=2,sticky=E+W)
34          circleButton = Button(fram, text = "Circle", \
35              command=circCommand)
36          circleButton.grid(row=1,column=1,columnspan=2)
37
38          def clickHandler(x,y):
39              print("In clickHandler")
40              if self.shapeSelection == circle:
41                  print("shape selection was circle")
42                  radius = radiusEnt.get()
43                  if radius.strip() == "":
44                      radius = 50
45                  else:
46                      radius = float(radius)
47                  shape = Circle(x,y,radius,edgeWidth=3, \
48                      color="red",outline="gray")
49                  shape.draw(aTurtle)
50                  screen.update()
51
52          screen.onclick(clickHandler)
53
54  def main():
55      app = DrawApp()
56      mainloop()
57
58  if __name__ == "__main__":
59      main()
```

The program in Example 7.7 is missing the *Circle* class which was defined in Example 7.4. The program waits for the *Circle* button to be pressed once. Then, after each mouse click, a circle is drawn on the ScrolledCanvas on the left side of the window.

Both a Circle and a Rectangle share a lot of common code. It makes sense for that common code to be in one base class that both classes inherit from. If a Shape class were defined that contained the shared code, then it would only have to be written once, which is a requirement of elegant code.

Example 7.8 Here is a Shape class that defines the code that is common to both Circles and Rectangles.

```
1   class Shape:
2       def __init__(self,x=0,y=0,color="transparent", \
3                    outline="black",width=1):
4           self.x = x
5           self.y = y
6           self.color = color
7           self.outline = outline
8           self.width = width
9
10      def setWidth(self,width):
11          self.width = width
```

```
12
13      def setFill(self,color):
14          self.color = color
15
16      def setOutline(self,color):
17          self.outline = color
18
19      def getX(self):
20          return self.x
21
22      def getY(self):
23          return self.y
```

With the Shape base class defined in Example 7.8 the definition of Circle can be simplified.

Example 7.9 Here is the code for the derived Circle class. Notice the call to *super*() below. *Super* refers to the superclass, in this case the *Shape* class. The superclass is the class that is above it in the type hierarchy. Using *super*() when referring to the superclass is a good idea because the code still works even if the type hierarchy is changed at some point in the future.

```
1   class Circle(Shape):
2       def __init__(self,x=0,y=0,radius=50,color="transparent", \
3                    outline="black",width=1):
4           super().__init__(x,y,color,outline,width)
5           self.radius = radius
6
7       def draw(self,turtle):
8           Shape.draw(self,turtle)
9           turtle.penup()
10          turtle.goto(self.x,self.y)
11          turtle.width(self.width)
12          if self.color != "transparent":
13              turtle.fillcolor(self.color)
14          turtle.color(self.outline)
15          turtle.fillcolor(self.color)
16          turtle.setheading(0)
17          turtle.forward(self.radius)
18          if self.color != "transparent":
19              turtle.begin_fill()
20          turtle.pendown()
21          for k in range(500):
22              radians = (2*math.pi)*(k/500.0)
23              turtle.goto(math.cos(radians)*self.radius+self.x, \
24                          math.sin(radians)*self.radius+self.y)
25          if self.color != "transparent":
26              turtle.end_fill()
27          turtle.penup()
28          turtle.goto(self.x,self.y)
29
30      def getRadius(self):
31          return self.radius
```

The Circle class still is the only class that will know how to draw a circle. And, of course, shapes don't have a radius in general. All the other code that isn't circle specific is now moved out of the Circle class.

Practice 7.4 Rewrite the Rectangle class so it inherits from the Shape class and use it in the draw program downloaded from the text's website.

7.3 A Bouncing Ball Example

A RawTurtle can move around the screen either with its pen up or its pen down. With its pen up, if we can imagine the turtle as something other than a little sprite, it can be essentially any object that we want it to be in a two dimensional world. The creators of the turtle graphics for Python realized this and added code so that we could change the turtle's picture to anything we would like. For instance, we might want to animate a bouncing ball. We can replace the turtle's sprite with an image of a ball.

Turtle graphics can do animation because it can be told to perform an action after an interval of time. A timer can be set in turtle graphics. When the timer goes off, the program can move the ball a little bit. If the interval between timer going off and moving the ball can be small enough that it happens several times a second, then to the human eye it will appear as if the ball is flying through the air.

A ball is a turtle. However, a turtle doesn't remember in which direction it is moving. It would be nice to have the ball remember the direction it is moving. At least somewhere in the program the ball's direction must be remembered and it makes sense for the ball to remember its own direction in an object-oriented design of the problem. Figure 7.5 depicts what a ball object should look like. A ball is a turtle, but it is a little more than just a turtle. Again, this is an example of inheritance.

With the ball inheriting from the RawTurtle class we'll automatically get all the functionality of a turtle. We can tell a ball to *goto* a location on the screen. We can access the x and y coordinate of the ball by calling the *xcor* and *ycor* methods. We can even change its shape so it looks like a ball. As we've seen, for the Ball class to inherit from the RawTurtle class, the derived Ball class must implement its own constructor and call the constructor of the base class.

Example 7.10 In Chap. 16 the Ball class inherits from the RawTurtle class. To create a Ball object we could write

```
ball = Ball(6.5,3.2)
```

This creates a ball object as shown in Fig. 7.5. Here is the Ball class code.

```
class Ball(RawTurtle):
    def __init__(self,cv,dx,dy):
        super().__init__(cv)

        self.penup()
        self.shape("soccerball.gif")
        self.dx = dx
        self.dy = dy

    def move(self):
        newx = self.xcor() + self.dx
        newy = self.ycor() + self.dy

        # some code goes here to make it bounce
        # off the walls.

        self.goto(newx,newy)
```

When we are using the ball object in Fig. 7.5 we refer to it using the *ball* reference. When we are in the Ball class we refer to the object using the *self* reference as described earlier in this chapter. In Fig. 7.5 the Turtle part of the object is greyed out. This is because the insides of the RawTurtle are available to us, but generally it is a bad idea to access the RawTurtle part of the object directly. Instead, we can use methods to access the RawTurtle part of the object when needed.

The constructor needs to initialize the RawTurtle part of the object as well as the Ball part of the object. To create a RawTurtle we could write *turtle* = *RawTurtle*(*cv*). However, writing this won't work to initialize the RawTurtle part of the object. A line of code like this would create a new RawTurtle object. Remember, a Ball

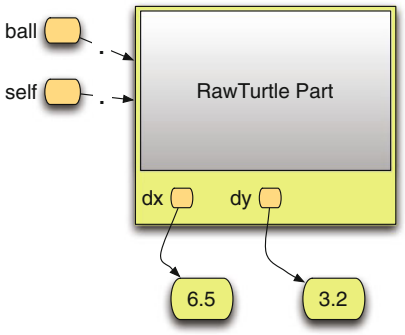

Fig. 7.5 A ball object

is a RawTurtle so we don't want to create a new RawTurtle object. Instead, we want to initialize the RawTurtle part of the Ball object. To do this, we explicitly call the RawTurtle constructor by writing *RawTurtle.__init__(self,cv)*. This calls the RawTurtle's constructor. In this case we call the constructor by writing the class name followed by a dot followed by the constructor's name *__init__*. Since self is a Ball *and* a RawTurtle, we pass self as the parameter to RawTurtle's constructor. This line of code initializes the RawTurtle part of the object. Then the Ball specific initialization occurs next.

The Ball class contains one more method, the move method. This is a new method not defined in the RawTurtle class. A Ball can move on the screen while a RawTurtle can not. A Ball moves by (dx,dy) each time the move method is called. The bouncing balls are animated by repeatedly calling the move method on each of the balls in the *ballList* defined in the main function of the program. Chapter 16 contains the complete code for the bouncing ball example.

7.4 Polymorphism

Polymorphism is a term used in object-oriented programming that means "many versions" or more than one version. When a subclass defines its own version of a method then the right version, either the subclass version or the base class version of the method, will be called depending on the type of object you have created. To best understand this it helps to look at an example.

Let's assume we wanted to modify the bouncing ball example so some balls bounce according to a simulated gravity instead of simply bouncing in space forever. It turns out this is very easy to do. We can have Ball objects bounce in space forever and GravityBall objects bounce according to a simulated gravity. Since GravityBalls are nearly the same as Balls we'll use inheritance to define the GravityBall class. The only real difference will be in the way the GravityBall moves when it is told to move.

Example 7.11 This code uses the Ball class and relies on polymorphism to get GravityBalls to bounce the right way.

```
class GravityBall(Ball):
    def __init__(self,cv,dx,dy):
        super().__init__(cv,dx,dy)

    def move(self):
        # Gravity's effect is -1/2 g t^2. Time is
        # estimated at 1/100 of a second for each
        # call to move.

        if abs(self.dy) < 0.2 and self.ycor() < 5:
            self.dy = 0
        else:
            self.dy = self.dy - 0.195

```

```
15            if abs(self.dx) < 0.2:
16                self.dx = 0
17            else:
18                # Friction reduces dx by a little bit
19                self.dx = 0.999 * self.dx
20
21            Ball.move(self)
```

Practice 7.5 Take the bouncing ball example and add the GravityBall class to it. Then, modify the program to create some GravityBalls and watch them bounce. The original Ball objects continue to bounce around as if they were in space. The GravityBall objects behave differently. Polymorphism makes this work. What is it about polymorphism that makes this work the way we want it to?

7.5 Getting Hooked on Python

A *hook* is a means by which one program allows another program to modify its behavior. The Python interpreter is a program that allows its behavior to be altered by means of certain hooks it makes available to programmers. Consider the Rational class described earlier in this chapter. With the definition you came up with (or the provided solution in practice Problem 7.1) we can create Rational numbers. However, we can't do much more than create them at the moment. Without some more code, our rational implementation doesn't really do us much good.

Example 7.12 Here is some code that creates a Rational number and prints it to the screen. When run, this program prints something like *< __main__.Rationalobjectat0x113bc70 >*. It prints the name of the module and the class and the value of the reference when printed to the screen.

```
1  class Rational:
2    def __init__(self,num=0,den=1):
3      self.num = num
4      self.den = den
5
6  def main():
7    x = Rational(4,5)
8    print(x)
9
10 if __name__ =="__main__":
11   main()
```

If we needed rational numbers in a program, it would be nice if they printed nicely when they were printed to the screen. This can be done using a hook in Python for string conversion. When an object is converted to a string, Python looks for the existence of the *__str__* method in the class. If this method exists, Python will use it to convert the object to a string representation. If this method exists in the class, then it must return a string representation of the object. The method must also have only one parameter, the self parameter.

Example 7.13 If this method is added to the Rational class definition in Example 7.12, then when the Rational 4/5 is printed, it prints as 4/5.

```
def __str__(self):
  return str(self.num)+"/"+str(self.den)
```

The addition of the *__str__* to the Rational class makes using rational numbers a bit easier because we can quickly convert it to a string when we want a nice representation of it. You can force the *__str__* method to be called by calling the *str* built-in function in Python. So, writing *str*(*x*) will force a string version of *x* to be constructed using the *__str__* method. The presence of the *__str__* method doesn't mean that rational numbers will *always* be converted to a string when printed. Sometimes, the Python interpreter isn't interested in producing a strictly human-readable presentation of an object. Sometimes a Python readable representation is more appropriate.

Example 7.14 Consider the following code. When Rational objects are in a list they do not print using the *__str__* method. Running this code prints [< *__main__.Rationalobjectat0x113bcd0* >, < *__main__.Rationalobjectat0x113bc70* >] to the screen.

```
1  def main():
2    x = Rational(4,5)
3    y = Rational(9,12)
4
5    lst = [x,y]
6    print(lst)
```

In Example 7.13 the *__str__* was added and rational numbers printed nicely, but Example 7.14 shows that the Python interpreter does not use *__str__* when printing a list of rationals. When printing a list, Python is producing a string representation of the list that would be suitable for Python to evaluate later to rebuild the list. If Python tried to read a number like 4/5 in the list, it would not know what to do with it. However, there is another hook that allows the programmer to determine the best representation of an object for Python's purposes.

Example 7.15 The *__repr__* method is a Python hook for producing a Python representation of an object. With the addition of the method below to the Rational class started in Example 7.12, Python will print [*Rational*(*4,5*), *Rational*(*9,12*)] when the code in Example 7.14 is executed.

```
def __repr__(self):
  return "Rational("+str(self.num)+","+str(self.den)+")"
```

So, what is the difference between converting to a string and converting to a Python representation? A string version of an object can be in whatever format the programmer determines is best. But, a Python representation should be in a format so that if the built-in Python function *eval* is called on it, it will evaluate to its original value. The *eval* function is given an expression contained in a string and evaluates the expression to produce the Python value contained in the string. The appropriate representation for most programmer-defined classes is to use the same form that is required to construct the object in the first place. To construct the rational number $4/5$ we had to write *Rational*(*4,5*). For the *eval* function to correctly evaluate a string containing a Rational, the eval function should be given a rational in the *Rational*(*numerator,denominator*) form, not the *numerator*/*denominator* form.

There is another Python hook that controls how sorting is performed in Python. For any type of object in Python, if there is a natural ordering to those objects, Python can sort a list of them.

Example 7.16 Here is some code that sorts a list of names, alphabetically. This code, when run, prints the list [*'Freeman', 'Gorman', 'Lee', 'Lie', 'Morgan'*] to the screen.

```
nameList = ["Lee", "Lie", "Gorman", "Freeman", "Morgan"]
nameList.sort()
print(nameList)
```

If we attempt to sort the list *lst* from Example 7.14, Python will complain with the following error message: *builtins.TypeError: unorderable types: Rational*() < *Rational*(). While we have an understanding of rational numbers, Python has no way of understanding that the class of Rational numbers represents an ordered collection of values. To tell Python that it is an ordered collection, we have to implement the *__lt__* method. To compare any two rational numbers, we must first make sure they have a common denominator. Once we have a common denominator, the numerator of the two rational numbers must be converted to units for the common denominator. It turns out we don't really need the common denominator at all. We just need the converted numerators. The *__lt__* method must return True if the object *self* references is less than the object that *other* references and it must return False otherwise.

Example 7.17 The following *__lt__* method, when added to the class in Example 7.12 converts the two numerators to their common denominator form so they can be compared.

```
1     def __lt__(self,other):
2       #commonDenominator = self.den * other.den
3       selfNum = self.numerator * other.den
4       otherNum = other.numerator * self.den
5       return selfNum < otherNum
```

Once the *__lt__* method of Example 7.17 is added to the Rational class, Python understands how to sort them. The sort function sorts a list in place as shown in Example 7.16. If *sort* is called on the list *lst* from Example 7.14, Python reorders the list so it contains [*Rational*(*9,12*), *Rational*(*4,5*)].

7.6 Review Questions

1. What is another name for a class in Python?
2. What is the relationship between classes and objects?
3. What is the purpose of the *__init__* method in a class definition?
4. Computer scientists say that objects have both *state* and *behavior*. What do *state* and *behavior* refer to in a class definition?
5. How do you create an object in Python?
6. In a class definition, when you see the word *self*, what does *self* refer to?
7. What is a superclass? Explain what the term means and give an example.
8. What is the benefit of inheritance in Python?
9. What does it mean for polymorphism to exist in a program? Why would you want this?
10. How do the *__str__* and the *__repr__* methods differ? Why are they both needed?
11. To be able to sort an ordered collection of your favorite type of objects, what method must be implemented on the objects?

7.7 Exercises

1. Go back to the original Reminder! program and redo it so that the Reminder! program contains a class called Reminder that replaces the parallel lists of reminders and notes with one list of reminders. This list should be a list of Reminder objects. A Reminder object keeps track of its x,y location on the screen. It also has some text that is provided when it is created. A Reminder must take care of creating the Text and Toplevel objects so a note can be displayed. Finally, the methods

defined on a Reminder include *undraw* (to withdraw the window), *getX* to return the X value of the window location, *getY* similarly gets the Y value of the window location. The *getText* method should return the text field. Finally, the *setDeleteHandler* should set the handler to be called when a reminder is deleted. Write this class and modify the Reminder! application to use this new class.
Here is an outline of the Reminder class definition. You need to finish defining it and alter the program to use it.

```
class Reminder:
    def __init__(self,x,y,text):
        ...

    def undraw(self):
        ...

    def getX(self):
            ...

    def getY(self):
            ...

    def getText(self):
            ...

    def setDeleteHandler(self,command):
        ...
```

Your job is to fill in the function definitions and then use the class in the Reminder! application.

2. Modify your address book program to use a class for address book cards. Call the new class AddressCard. An address card contains all the information for an address book entry including last and first name, street, city, state, zip, phone, and mobile phone number. To use the AddressCard class you need to modify the program so it stores all AddressCards in a list. The program should read all the addresses when it starts and make one AddressCard object for each address in the file. You will also write all the cards in the list to a file when the program terminates. Look at the code in Chap. 15 to see how this can be done.
You will want to include three hook methods in your AddressCard class. The *__str__* method should be included to convert an AddressCard to a string. To do this you will want to return a string representation of the object as discussed in the chapter. The AddressCard entry should convert to a string as follows:

```
Sophus Lie
Abel Avenue
Lavanger, Norway 554433
555-555-5555
444-444-4444
```

Your *__str__* method should return a string that looks just like this. When you print your addresses to the file when the application closes, you can use the *str* function to convert each AddressCard object to a string. Don't forget the newline characters at the end of each line.
The second special method is the *__lt__* method. This method compares two AddressCard objects as described for Rationals in the chapter. Your *__lt__* method

should return True if the last name, first name of self is less than the last name, first name of the other AddressCard.
A third special method is the *__eq__* method. This method compares two AddressCard objects and is used by the *index* method on lists. If *self* is equal to *other* then *True* should be returned. If *self* is not equal to *other* then *False* should be returned. Here is how you might write this function.

```
class AddressCard:

    ....

    # This method provides a means of comparing
    # the current object (i.e. self) with another
    # object. It is used by the index method on lists
    # to discover if an object in a list "equals" the
    # object being searched for by the index method.

    def __eq__(self,other):
        if type(other) != type(self):
            raise "Invalid Comparison"

        if self.last+","+self.first == \
            other.last+","+other.first:

            return True

        return False
```

Each of the event handlers must be rewritten to use the new list of addresses. For instance, here is how the *Find* event handler might be written to use the index method on lists that is now possible with the definition of the *__eq__* method.

```
    def findAddress():
        print("Find")

        card = AddressCard(lname.get().strip(), \
            fname.get().strip(),street.get().strip(), \
            city.get().strip(),state.get().strip(), \
            zip.get().strip(), phone.get().strip(), \
            mobile.get().strip())

        try:
            j = addresses.index(card)
            card = addresses[j]

            street.set(card.getStreet())
            city.set(card.getCity())
            state.set(card.getState())
            zip.set(card.getZip())
            phone.set(card.getPhone())
            mobile.set(card.getMobile())
            return

        except:
            True

        messagebox.showwarning("Not Found", \
            "The entry was not found!")
```

Finally, you should use the *list* sort method to keep the address book sorted at all times.

3. In this exercise you are to implement a game of Blackjack using the turtle package. Blackjack is a simple game with simple rules. In this exercise you get practice using Object-Oriented Programming to implement a fairly complex program.

   Rules of the Game

   Blackjack is played by dealing two cards to each player and the dealer. The player's cards are face up. The dealer's first card is face down and the second is face up.

   The goal is to get to 21 points. Each face card is worth 10 points. The Ace is worth 1 or 11 points depending on which is better for your hand. All other cards are worth their face value.

   The player bets first. Then he/she asks for cards (hits) until they are satisfied with their score or they go over. If they have not gone over, the dealer then draws cards until the dealer hand is 17 or over. If the dealer goes over 21, the player wins. Otherwise, the player wins if his/her score is greater than the dealer's score.

   If the player gets a blackjack (21 with only two cards) then the player gets paid at a 3:2 ratio. Otherwise it is a 1:1 ratio payback.

   Writing the Game

   You should write this game incrementally. That means, write a little bit and test that little bit before going on. You don't want to debug this whole program after writing all the code.

   You will need to implement a Card class. A Card object can inherit from RawTurtle. When you create a Card object you will want to give it an image for the front and back. The images can be downloaded from the text website. Download the cards.zip file, and then unzip it in the same folder where you will write your program. The cards folder should be a subfolder of the folder where you write your program.

   The card images are named 1.gif, 2.gif, and so on. The back image is labeled back.gif. Images 1, 2, 3, 4 are the Aces. Images 5, 6, 7, 8 are the Kings and so on. To get the correct rank for a card you can use the formula $14 - val/4$ where val is the value of the card name. If the formula determines the rank is 14 it should be changed to 11. Ranks from 10–13 should be changed to 10.

   The Card class will have at least four methods. You may want to define more. Here is a suggestion for the methods you should write.

   - isFaceDown—This method returns true if the card is face down. It returns false if the card is face up.
   - setFaceDown—This method sets the Turtle shape to be the back of the card and remembers that it is now face down.
   - setFaceUp—This method sets the Turtle shape to be the face of the card and remembers that the card is now face up.
   - getBlackJackRank—This method returns the Blackjack rank of the card.

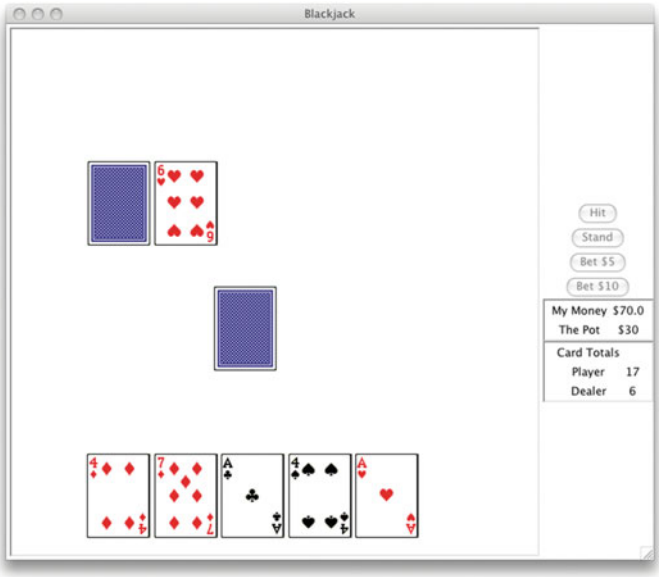

Fig. 7.6 A Blackjack hand

The main part of the program is placing buttons on the screen and handling the different button presses. Figure 7.6 shows what the application might look like during the playing of a hand. Figure 7.7 shows what the application might display at the end of that hand. Message boxes can be used to display the outcome of a hand.

4. Complete the Asteroids game available on the text web site as shown in Fig. 7.8. The Asteroids video game was originally designed and written by Atari. It was released to the public in 1979. In the game the player controls a spaceship that navigates space and blows up asteroids by shooting at them.
   When an asteroid is hit, the player scores points and the asteroid splits into two smaller asteroids. The largest asteroids are worth 20 points. Each medium asteroid is worth 50 points. The smallest asteroids are worth 100 points each. When the spaceship hits a small asteroid it is obliterated into dust and it disappears completely from the game.
   If an asteroid collides with the spaceship, the spaceship is destroyed, the asteroid that collided with it is destroyed (resulting in no points) and the player gets a new spaceship. The game starts with four spaceships total (the original game started with only three).
   Code is available on the text's web site. The downloadable code makes the ship turn left when 4 is pressed. The ship will also move forward when 5 is pressed. Complete the program by implementing the game as described above. Some lessons are available on the text's web site that will guide you through many of the additions to the program described here. To make the game a little more

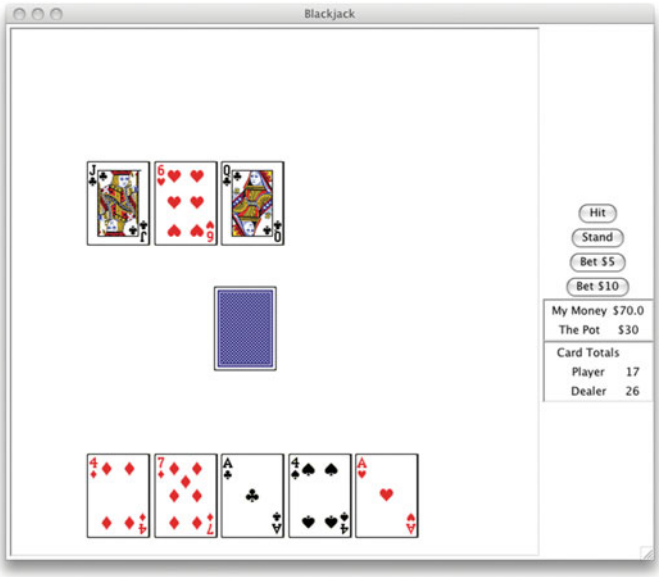

Fig. 7.7 The end of a Blackjack hand

interesting you should add one new level to this program. The second level should have 7 asteroids instead of 5 and you should get one more life if you have less than 4 when level 2 starts.

5. In Chap. 4, XML documents were introduced. The example in that chapter was of drawing a picture contained in an XML file. To do this, several parallel lists were constructed to hold the data of the XML file. However, there were lots of *None* values placed in the lists because not all attributes applied to all graphics commands.

   A much better way of organizing the XML data would be to create a class for each different kind of graphics command. So a BeginFillCommand class would contain just the color attribute needed for the BeginFill graphics command. Likewise, the class associated with each different type of command would hold the attributes needed just for that command. Then, a draw method could be written for each class that draws or uses a turtle for the desired side-effiect. Each draw method should be passed a turtle. The BeginFillCommand's draw method would use the turtle to set the *fillcolor* and then would invoke the *begin_fill* turtle method.

   Rewrite the XML drawing program from Chap. 4 by defining a class for each type of graphics command along with a draw method for each of them that given a turtle draws or otherwise has the desired side-effect. Have the program read the XML file and create *one* list of these graphics command objects. Then use a loop to iterate through these commands, drawing each of them to the screen. Once completed you will have eliminated all the parallel lists from the program and written it with a much more object-oriented approach.

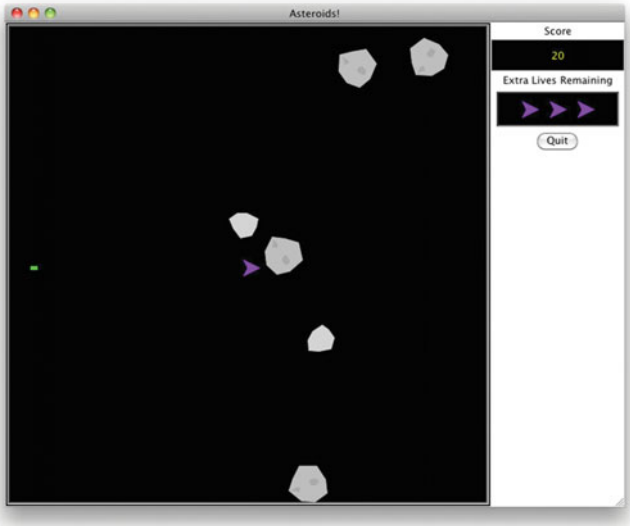

Fig. 7.8 Asteroids!

7.8 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.

7.8.1 Solutions to Practice Problem 7.1

A numerator and denominator are needed.

```
class Rational:
    def __init__(self,num=0,den=1):
        self.num = num
        self.den = den
```

7.8.2 Solutions to Practice Problem 7.2

```
class Rectangle(Shape):
    def __init__(self,x,y,width,height,color="transparent", \
                 outline="black",edgeWidth=1):
        self.x = x
        self.y = y
        self.color = color
        self.outline = outline
        self.edgeWidth = edgeWidth
```

```
9           self.width = width
10          self.height = height
```

7.8.3 Solutions to Practice Problem 7.3

```
1       def draw(self,turtle):
2           turtle.penup()
3           turtle.goto(self.x,self.y)
4           turtle.setheading(0)
5           turtle.pendown()
6           turtle.width(self.edgeWidth)
7           turtle.color(self.outline)
8           turtle.fillcolor(self.color)
9           if self.color != "transparent":
10              turtle.begin_fill()
11          turtle.pendown()
12          turtle.forward(self.width)
13          turtle.left(90)
14          turtle.forward(self.height)
15          turtle.left(90)
16          turtle.forward(self.width)
17          turtle.left(90)
18          turtle.forward(self.height)
19          turtle.left(90)
20          if self.color != "transparent":
21              turtle.end_fill()
22          turtle.penup()
```

7.8.4 Solutions to Practice Problem 7.4

Download code and try it out. Here is the Rectangle class in case you had trouble with defining it.

```
1   class Rectangle(Shape):
2       def __init__(self,x,y,width,height,color="transparent", \
3                    outline="black",edgeWidth=1):
4           super().__init__(x,y,color,outline,edgeWidth)
5
6           self.width = width
7           self.height = height
8
9       def draw(self,turtle):
10          turtle.penup()
11          turtle.goto(self.x,self.y)
12          turtle.setheading(0)
13          turtle.pendown()
14          turtle.width(self.edgeWidth)
15          turtle.color(self.outline)
16          turtle.fillcolor(self.color)
17          if self.color != "transparent":
18              turtle.begin_fill()
19          turtle.pendown()
20          turtle.forward(self.width)
21          turtle.left(90)
22          turtle.forward(self.height)
23          turtle.left(90)
24          turtle.forward(self.width)
25          turtle.left(90)
26          turtle.forward(self.height)
27          turtle.left(90)
```

```
28            if self.color != "transparent":
29                turtle.end_fill()
30            turtle.penup()
```

7.8.5 Solutions to Practice Problem 7.5

Create some GravityBall objects and add them to the ballList. That's all that needs to be done to have gravity balls and regular balls bouncing around with each other. The object on the left hand side of the dot in the ball *ball.move* is where polymorphism is at work. If *ball* is pointing to a Ball object, it behaves as a Ball would. If *ball* is pointing to a GravityBall object, then *ball.move* is the GravityBall move method. It's not the name on the left hand side of the dot, its the object that the name refers to that controls which methods are called.