Chapter 5 Objects and Graphics

Objectives

- To understand the concept of objects and how they can be used to simplify programs.
- To be familiar with the various objects available in the graphics library.
- To be able to create objects in programs and call appropriate methods to perform graphical computations.
- To understand the fundamental concepts of computer graphics, especially the role of coordinate systems and coordinate transformations.
- To understand how to work with both mouse and text-based input in a graphical programming context.
- To be able to write simple interactive graphics programs using the graphics library.

5.1 Overview

So far we have been writing programs that use the built-in Python data types for numbers and strings. We saw that each data type could represent a certain set of values, and each had a set of associated operations. Basically, we viewed the data as passive entities that were manipulated and combined via active operations. This is a traditional way to view computation. To build complex systems,

however, it helps to take a richer view of the relationship between data and operations.

Most modern computer programs are built using an *object-oriented* (OO) approach. Object orientation is not easily defined. It encompasses a number of principles for designing and implementing software, principles that we will return to numerous times throughout the course of this book. This chapter provides a basic introduction to object concepts by way of some computer graphics.

Graphical programming is a lot of fun and provides a great vehicle for learning about objects. In the process, you will also learn the principles of computer graphics that underlie many modern computer applications. Most of the applications that you are familiar with probably have a so-called *Graphical User Interface* (GUI) that provides visual elements like windows, icons (representative pictures), buttons and menus.

Interactive graphics programming can be very complicated; entire textbooks are devoted to the intricacies of graphics and graphical interfaces. Industrial-strength GUI applications are usually developed using a dedicated graphics programming framework. Python comes with its own standard GUI module called *Tkinter*. As GUI frameworks go, Tkinter is one of the simplest to use, and Python is a great language for developing real-world GUIs. Still, at this point in your programming career, it would be a challenge to learn the intricacies of any GUI framework, and doing so would not contribute much to the main objectives of this chapter, which are to introduce you to objects and the fundamental principles of computer graphics.

To make learning these basic concepts easier, we will use a graphics library (`graphics.py`) specifically written for use with this textbook. This library is a wrapper around Tkinter that makes it more suitable for beginning programmers. It is freely available as a Python module file[1] and you are welcome to use it as you see fit. Eventually, you may want to study the code for the library itself as a stepping stone to learning how to program directly in Tkinter.

5.2 The Object of Objects

The basic idea of object-oriented development is to view a complex system as the interaction of simpler *objects*. The word *objects* is being used here in a specific technical sense. Part of the challenge of OO programming is figuring out the

[1] See Appendix B for information on how to obtain the graphics library and other supporting materials for this book.

vocabulary. You can think of an OO object as a sort of active data type that combines both data and operations. To put it simply, objects *know stuff* (they contain data), and they *can do stuff* (they have operations). Objects interact by sending each other messages. A message is simply a request for an object to perform one of its operations.

Consider a simple example. Suppose we want to develop a data processing system for a college or university. We will need to keep track of considerable information. For starters, we must keep records on the students who attend the school. Each student could be represented in the program as an object. A student object would contain certain data such as name, ID number, courses taken, campus address, home address, GPA, etc. Each student object would also be able to respond to certain requests. For example, to send out a mailing, we would need to print an address for each student. This task might be handled by a `printCampusAddress` operation. When a particular student object is sent the `printCampusAddress` message, it prints out its own address. To print out all the addresses, a program would loop through the collection of student objects and send each one in turn the `printCampusAddress` message.

Objects may refer to other objects. In our example, each course in the college might also be represented by an object. Course objects would know things such as who the instructor is, what students are in the course, what the prerequisites are, and when and where the course meets. One example operation might be `addStudent`, which causes a student to be enrolled in the course. The student being enrolled would be represented by the appropriate student object. Instructors would be another kind of object, as well as rooms, and even times. You can see how successive refinement of these ideas could lead to a rather sophisticated model of the information structure of the college.

As a beginning programmer, you're probably not yet ready to tackle a college information system. For now, we'll study objects in the context of some simple graphics programming.

5.3 Simple Graphics Programming

In order to run the graphical programs and examples in this chapter (and the rest of the book), you will need a copy of the file `graphics.py` that is supplied with the supplemental materials. Using the graphics library is as easy as placing a copy of the `graphics.py` file in the same folder as your graphics program(s). Alternatively, you can place it in a system directory where other Python libraries are stored so that it can be used from any folder on the system.

The graphics library makes it easy to experiment with graphics interactively and write simple graphics programs. As you do, you will be learning principles of object-oriented programming and computer graphics that can be applied in more sophisticated graphical programming environments. The details of the `graphics` module will be explored in later sections. Here we'll concentrate on a basic hands-on introduction to whet your appetite.

As usual, the best way to start learning new concepts is to roll up your sleeves and try out some examples. The first step is to import the graphics module. Assuming you have placed `graphics.py` in an appropriate place, you can import the graphics commands into an interactive Python session.

```
>>> import graphics
```

Next we need to create a place on the screen where the graphics will appear. That place is a *graphics window* or `GraphWin`, which is provided by the `graphics` module.

```
>>> win = graphics.GraphWin()
```

This command creates a new window on the screen. The window will have the title "Graphics Window." The `GraphWin` may overlap your Python interpreter window, so you might have to resize the Python window to make both fully visible. Figure 5.1 shows an example screen view.

The `GraphWin` is an object, and we have assigned it to the variable called `win`. We can manipulate the window object through this variable, similar to the way that file objects are manipulated through file variables. For example, when we are finished with a window, we can destroy it. This is done by issuing the `close` command.

```
>>> win.close()
```

Typing this command causes the window to vanish from the screen.

We will be working with quite a few commands from the `graphics` library, and it gets tedious having to type the `graphics.` notation every time we use one. Python has an alternative form of `import` that can help out.

```
from graphics import *
```

The `from` statement allows you to load specific definitions from a library module. You can either list the names of definitions to be imported or use an asterisk, as shown, to import everything defined in the module. The imported commands become directly available without having to preface them with the module name. After doing this import, we can create a `GraphWin` more simply.

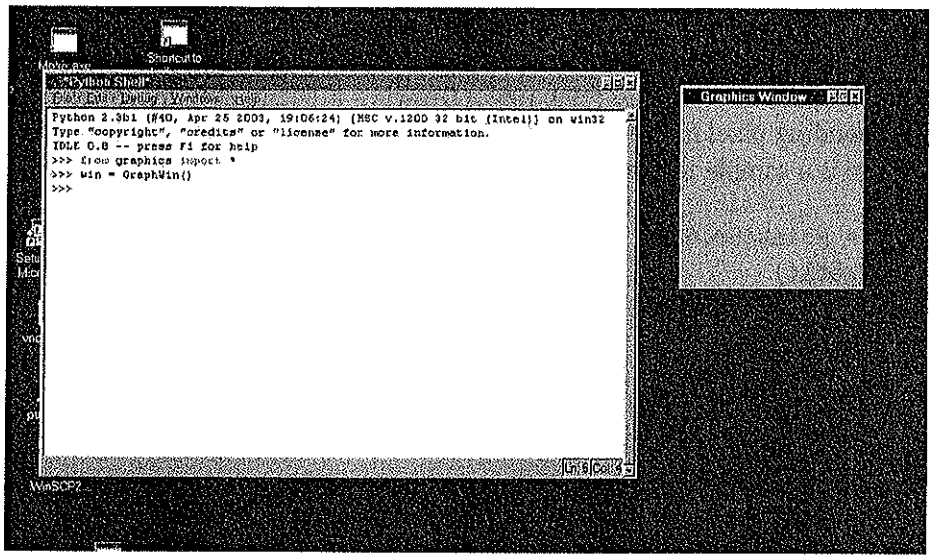

Figure 5.1: Screen shot with a Python window and a GraphWin.

```
win = GraphWin()
```

All of the rest of the graphics examples will assume that the entire `graphics` module has been imported using `from`.

Let's try our hand at some drawing. A graphics window is actually a collection of tiny points called *pixels* (short for picture elements). By controlling the color of each pixel, we control what is displayed in the window. By default, a `GraphWin` is 200 pixels tall and 200 pixels wide. That means there are 40,000 pixels in the `GraphWin`. Drawing a picture by assigning a color to each individual pixel would be a daunting challenge. Instead, we will rely on a library of graphical objects. Each type of object does its own bookkeeping and knows how to draw itself into a `GraphWin`.

The simplest object in the `graphics` module is a `Point`. In geometry, a point is a location in space. A point is located by reference to a coordinate system. Our graphics object `Point` is similar; it can represent a location in a `GraphWin`. We define a point by supplying x and y coordinates (x, y). The x value represents the horizontal location of the point, and the y value represents the vertical.

Traditionally, graphics programmers locate the point $(0, 0)$ in the upper-left

corner of the window. Thus x values increase from left to right, and y values increase from top to bottom. In the default 200 x 200 `GraphWin`, the lower-right corner has the coordinates $(199, 199)$. Drawing a `Point` sets the color of the corresponding pixel in the `GraphWin`. The default color for drawing is black.

Here is a sample interaction with Python illustrating the use of `Points`:

```
>>> p = Point(50,60)
>>> p.getX()
50
>>> p.getY()
60
>>> win = GraphWin()
>>> p.draw(win)
>>> p2 = Point(140,100)
>>> p2.draw(win)
```

The first line creates a `Point` located at $(50, 60)$. After the `Point` has been created, its coordinate values can be accessed by the operations `getX` and `getY`. A `Point` is drawn into a window using the `draw` operation. In this example, two different point objects (`p` and `p2`) are created and drawn into the `GraphWin` called `win`. Figure 5.2 shows the resulting graphical output.

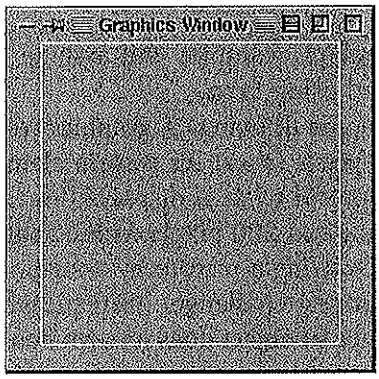

Figure 5.2: Graphics window with two points drawn.

In addition to points, the graphics library contains commands for drawing lines, circles, rectangles, ovals, polygons and text. Each of these objects is cre-

ated and drawn in a similar fashion. Here is a sample interaction to draw various shapes into a GraphWin:

```
>>> #### Open a graphics window
>>> win = GraphWin('Shapes')
>>> #### Draw a red circle centered at point (100,100) with radius 30
>>> center = Point(100,100)
>>> circ = Circle(center, 30)
>>> circ.setFill('red')
>>> circ.draw(win)
>>> #### Put a textual label in the center of the circle
>>> label = Text(center, "Red Circle")
>>> label.draw(win)
>>> #### Draw a square using a Rectangle object
>>> rect = Rectangle(Point(30,30), Point(70,70))
>>> rect.draw(win)
>>> #### Draw a line segment using a Line object
>>> line = Line(Point(20,30), Point(180, 165))
>>> line.draw(win)
>>> #### Draw an oval using the Oval object
>>> oval = Oval(Point(20,150), Point(180,199))
>>> oval.draw(win)
```

Try to figure out what each of these statements does. If you type them in as shown, the final result will look like Figure 5.3.

5.4 Using Graphical Objects

Some of the examples in the above interactions may look a bit strange to you. To really understand the graphics module, we need to take an object-oriented point of view. Remember, objects combine data with operations. Computation is performed by asking an object to carry out one of its operations. In order to make use of objects, you need to know how to create them and how to request operations.

In the interactive examples above, we manipulated several different kinds of objects: GraphWin, Point, Circle, Oval, Line, Text, and Rectangle. These are examples of *classes*. Every object is an *instance* of some class, and the class describes the properties the instance will have.

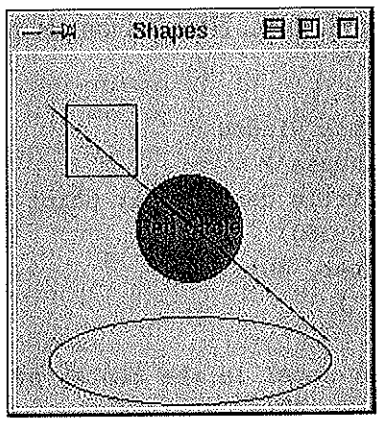

Figure 5.3: Various shapes from the `graphics` module.

Borrowing a biological metaphor, when we say that Fido is a dog, we are actually saying that Fido is a specific individual in the larger class of all dogs. In OO terminology, Fido is an instance of the dog class. Because Fido is an instance of this class, we expect certain things. Fido has four legs, a tail, a cold, wet nose and he barks. If Rex is a dog, we expect that he will have similar properties, even though Fido and Rex may differ in specific details such as size or color.

The same ideas hold for our computational objects. We can create two separate instances of `Point`, say `p` and `p2`. Each of these points has an x and y value, and they both support the same set of operations like `getX` and `draw`. These properties hold because the objects are `Points`. However, different instances can vary in specific details such as the values of their coordinates.

To create a new instance of a class, we use a special operation called a *constructor*. A call to a constructor is an expression that creates a brand new object. The general form is as follows:

```
<class-name>(<param1>, <param2>, ...)
```

Here `<class-name>` is the name of the class that we want to create a new instance of, e.g., `Circle` or `Point`. The expressions in the parentheses are any parameters that are required to initialize the object. The number and type of the parameters depends on the class. A `Point` requires two numeric values, while a `GraphWin` can be constructed without any parameters. Often, a constructor is used on the right side of an assignment statement, and the resulting

object is immediately assigned to a variable on the left side that is then used to manipulate the object.

To take a concrete example, let's look at what happens when we create a graphical point. Here is a constructor statement from the interactive example above.

```
p = Point(50,60)
```

The constructor for the `Point` class requires two parameters giving the x and y coordinates for the new point. These values are stored as *instance variables* inside of the object. In this case, Python creates an instance of `Point` having an x value of 50 and a y value of 60. The resulting point is then assigned to the variable `p`. A conceptual diagram of the result is shown in Figure 5.4. Note that, in this diagram as well as similar ones later on, only the most salient details are shown. `Points` also contain other information such as their color and which window (if any) they are drawn in. Most of this information is set to default values when the `Point` is created.

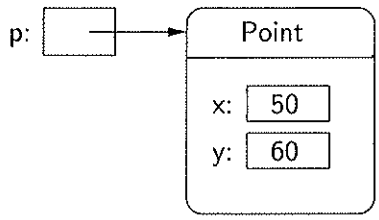

Figure 5.4: Conceptual picture of the result of `p = Point(50,60)`. The variable `p` refers to a freshly created `Point` having the given coordinates.

To perform an operation on an object, we send the object a message. The set of messages that an object responds to are called the *methods* of the object. You can think of methods as functions that live inside of the object. A method is invoked using dot-notation.

```
<object>.<method-name>(<param1>, <param2>, ...)
```

The number and type of the parameters is determined by the method being used. Some methods require no parameters at all. You can find numerous examples of method invocation in the interactive examples above.

As examples of parameterless methods, consider these two expressions:

```
p.getX()
p.getY()
```

The `getX` and `getY` methods return the x and y values of a point, respectively. Methods such as these are sometimes called *accessors*, because they allow us to access information from the instance variables of the object.

Other methods change the values of an object's instance variables, hence changing the *state* of the object. All of the graphical objects have a `move` method. Here is a specification:

`move(dx, dy)`: Moves the object `dx` units in the x direction and `dy` units in the y direction.

To move the point `p` to the right 10 units, we could use this statement.

```
p.move(10,0)
```

This changes the x instance variable of `p` by adding 10 units. If the point is currently drawn in a `GraphWin`, `move` will also take care of erasing the old image and drawing it in its new position. Methods that change the state of an object are sometimes called *mutators*.

The `move` method must be supplied with two simple numeric parameters indicating the distance to move the object along each dimension. Some methods require parameters that are themselves complex objects. For example, drawing a `Circle` into a `GraphWin` involves two objects. Let's examine a sequence of commands that does this.

```
circ = Circle(Point(100,100), 30)
win = GraphWin()
circ.draw(win)
```

The first line creates a `Circle` with a center located at the `Point` $(100, 100)$ and a radius of 30. Notice that we used the `Point` constructor to create a location for the first parameter to the `Circle` constructor. The second line creates a `GraphWin`. Do you see what is happening in the third line? This is a request for the `Circle` object `circ` to draw itself into the `GraphWin` object `win`. The visible effect of this statement is a circle in the `GraphWin` centered at $(100, 100)$ and having a radius of 30. Behind the scenes, a lot more is happening.

Remember, the `draw` method lives inside the `circ` object. Using information about the center and radius of the circle from the instance variables, the `draw` method issues an appropriate sequence of low-level drawing commands

(a sequence of method invocations) to the `GraphWin`. A conceptual picture of the interactions among the `Point`, `Circle` and `GraphWin` objects is shown in Figure 5.5. Fortunately, we don't usually have to worry about these kinds of details; they're all taken care of by the graphical objects. We just create objects, call the appropriate methods, and let them do the work. That's the power of object-oriented programming.

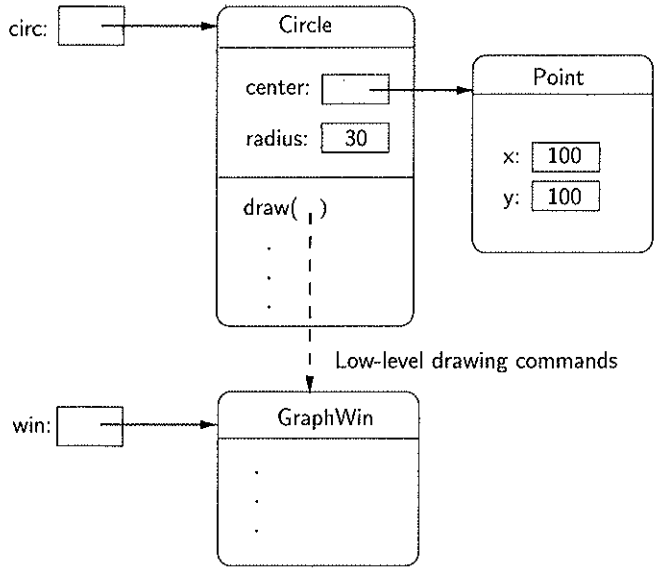

Figure 5.5: Object interactions to draw a circle.

There is one subtle "gotcha" that you need to keep in mind when using objects. It is possible for two different variables to refer to exactly the same object; changes made to the object through one variable will also be visible to the other. Suppose we are trying to write a sequence of code that draws a smiley face. We want to create two eyes that are 20 units apart. Here is a sequence of code intended to draw the eyes.

```
## Incorrect way to create two circles.
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye
```

```
rightEye.move(20,0)
```

The basic idea is to create the left eye and then copy that into a right eye which is then moved over 20 units.

This doesn't work. The problem here is that only one `Circle` object is created. The assignment

```
rightEye = leftEye
```

simply makes `rightEye` refer to the very same circle as `leftEye`. Figure 5.6 shows the situation. When the `Circle` is moved in the last line of code, both `rightEye` and `leftEye` refer to it in its new location on the right side. This situation where two variables refer to the same object is called *aliasing*, and it can sometimes produce rather unexpected results.

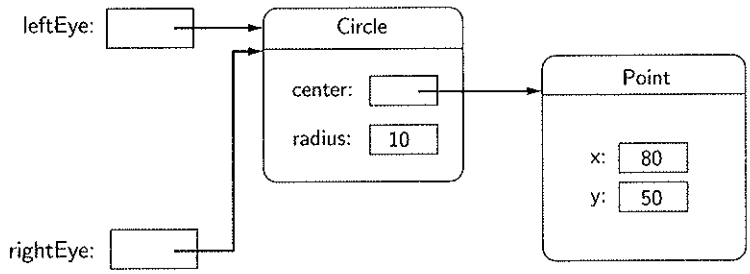

Figure 5.6: Variables `leftEye` and `rightEye` are aliases.

One solution to this problem would be to create a separate circle for each eye.

```
## A correct way to create two circles.
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = Circle(Point(100, 50), 5)
rightEye.setFill('yellow')
rightEye.setOutline('red')
```

This will certainly work, but it's cumbersome. We had to write duplicated code for the two eyes. That's easy to do using a "cut and paste" approach, but it's not

very elegant. If we decide to change the appearance of the eyes, we will have to be sure to make the changes in two places.

The `graphics` library provides a better solution; all graphical objects support a `clone` method that makes a copy of the object. Using `clone`, we can rescue the original approach.

```
## Correct way to create two circles, using clone.
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye.clone() # rightEye is an exact copy of the left
rightEye.move(20,0)
```

Strategic use of cloning can make some graphics tasks much easier.