Chapter 13

Turtle Graphics

13.1 Introduction

Graphical User Interfaces (GUI's) provide a rich environment in which information can be exchanged between a user and the computer. GUI's are not limited to simply displaying text and reading text from the keyboard. GUI's enable users to control the behavior of a program by performing actions such as using the mouse to drag or click on graphical objects. GUI's can make using programs much more intuitive and easier to learn since they provide users with immediate visual feedback that shows the effects of their actions.

There are many Python packages that can be used to create graphics and GUI's. Two graphics modules, called turtle and tkinter, come as a part of Python's standard library. tkinter is primarily designed for creating GUI's. In fact, IDLE is built using tkinter. However, we will focus on the turtle module that is primarily used as a simple graphics package but can also be used to create simple GUI's.

The turtle module is an implementation of turtle graphics and uses tkinter for the creation of the underlying graphics. Turtle graphics dates back to the 1960's and was part of the Logo programming language.[1] This chapter provides an introduction to using the graphics capabilities of the turtle module and demonstrates the creation of simple images and simple GUI's for games and applications. We will not delve into the details of building and designing GUI's, but many of the skills developed here can be applied to more complicated GUI designs if you wish to pursue that in the future. In addition to helping you gain practical programming skills, learning to use turtle graphics is fun and it enables you to use Python to be visually creative!

13.2 Turtle Basics

Among other things, the methods in the turtle module allow us to draw images. The idea behind the turtle part of "turtle graphics" is based on a metaphor. Imagine you have a turtle on a canvas that is holding a pen. The pen can be either up (not touching the canvas) or down (touching the canvas). Now think of the turtle as a robot that you can control by issuing commands. When the

---

From the file: `turtle-graphics.tex`

[1]See **en.wikipedia.org/wiki/Turtle_graphics**

pen it holds is down, the turtle leaves a trail when you tell it to move to a new location. When the pen is up, the turtle moves to a new position but no trail is left. In addition to position, the turtle also has a heading, i.e., a direction, of forward movement. The turtle module provides commands that can set the turtle's position and heading, control its forward and backward movement, specify the type of pen it is holding, etc. By controlling the movement and orientation of the turtle as well as the pen it is holding, you can create drawings from the trails the turtle leaves.

13.2.1   Importing Turtle Graphics

In order to start using turtle graphics, we need to import the turtle module. Start Python/IDLE and type the following:

Listing 13.1 Importing the turtle module.

```
>>> import turtle as t
```

This imports the turtle module using the identifier t. By importing the module this way we access the methods within the module using t.<object> instead of turtle.<object>. To ensure that the module was properly imported, use the dir() function as shown in Listing 13.2.

Listing 13.2 Using dir() to view the turtle module's methods and attributes.

```
1  >>> dir(t)
2  ['Canvas', 'Pen', 'RawPen', 'RawTurtle', 'Screen', 'ScrolledCanvas',
3      'Shape', 'TK', 'TNavigator', 'TPen', 'Tbuffer', 'Terminator',
4      'Turtle', 'TurtleGraphicsError', 'TurtleScreen', 'TurtleScreenBase',
5      'Vec2D', '_CFG', '_LANGUAGE', '_Root', '_Screen', '_TurtleImage',
6      '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
7          ...   <<MANY LINES OF OUTPUT DELETED>>
8      'window_width', 'write', 'write_docstringdict', 'xcor', 'ycor']
```

The list returned by the dir() function in Listing 13.2 has been truncated—in all, there are 173 items in this list. To learn more about any of these attributes or methods, you can use the help() function. For example, to learn about the forward() method, enter the statement shown in line 1 of Listing 13.3.

Listing 13.3 Learning about the forward() method using help().

```
1  >>> help(t.forward)
2  Help on function forward in module turtle:
3
4  forward(distance)
5      Move the turtle forward by the specified distance.
6
```

```
7    Aliases: forward | fd
8
9    Argument:
10   distance -- a number (integer or float)
11
12   Move the turtle forward by the specified distance, in the direction
13   the turtle is headed.
14              ...
15   <<REMAINING OUTPUT DELETED>>
```

From this we learn, as shown in lines 12 and 13, that this method moves "the turtle forward by the specified distance, in the direction the turtle is headed." We also learn that there is a shorter name for this method: `fd()`.

13.2.2 Your First Drawing

Let's begin by telling our turtle to draw a line. Try entering the command shown in Listing 13.4.

Listing 13.4 Drawing a line with a length of 100 units.

```
>>> t.fd(100)
```

As shown in Fig. 13.1, a graphics window should appear in which you see a small arrow 100 units to the right of the center of the window.[2] A thin black line is drawn from the center of the window to the tail of the arrow. The arrow represents our "turtle" and the direction the arrow is pointing indicates the current heading. The `fd()` method is a shorthand for the `forward()` method—the two methods are identical. `fd()` takes one integer argument that specifies the number of units you want to move the turtle forward in the direction of the current heading.[3] If you provide a negative argument, the turtle moves backwards the specified amount. Alternatively, to move backward one can call either `backward()`, `back()`, or `bk()`.

The default shape for our turtle is an arrow but if we wanted to have it look like a turtle we could type the command shown in Listing 13.5.

Listing 13.5 Changing the shape of the turtle.

```
>>> t.shape("turtle")
```

This replaces the arrow with a small turtle. We can change the shape of our turtle to a number of other built in shapes using the `shape()` method. We can also create custom shapes although we won't cover that here.

---

[2]When it first opens, this window may appear behind previously opened windows. So, you may have to search for it.

[3]By default the units correspond to pixels, i.e., individual picture-elements or dots on your screen, but one can reset the coordinates so that the units can correspond to whatever is most convenient to generate the desired image.
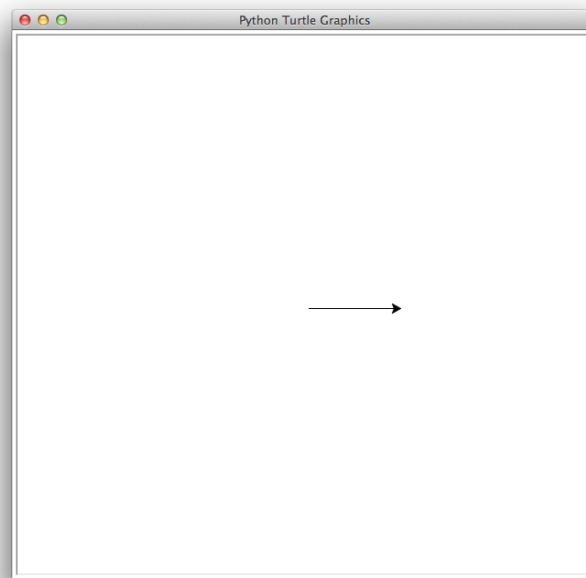
Figure 13.1: A line of length 100 produced by the `fd()` method.

Even though our turtle's shape appears on the graphics window, the turtle is not truly part of our drawing. The shape of the turtle is there to help you see the turtle's current position and heading, but you need to issue other commands, such as `fd()`, to create a drawing. If you have created a masterpiece and you no longer want to see the turtle in your graphics window, you can enter the command shown in Listing 13.6.

Listing 13.6 Command to hide the turtle's shape from the screen.

```
>>> t.hideturtle()
```

This hides the image that currently represents the turtle. In fact, you can continue to create lines even when the turtle's shape is hidden, but you will not be able to see the turtle's current position nor its heading. If you want to see the turtle again, simply issue the command shown in Listing 13.7.

Listing 13.7 Making the turtle visible.

```
>>> t.showturtle()
```

The turtle's heading can be controlled using one of three methods: `left()`, `right()`, and `setheading()`; or the shorter aliases of `lt()`, `rt()`, and `seth()`, respectively. `left()` and `right()` turn the turtle either to the left or right, respectively, by the number of degrees given as the argument. These turns are relative to the turtle's current heading. So, for example, `left(45)` causes the turtle to turn 45 degrees to the left. On the other hand, `setheading()` and `seth()`
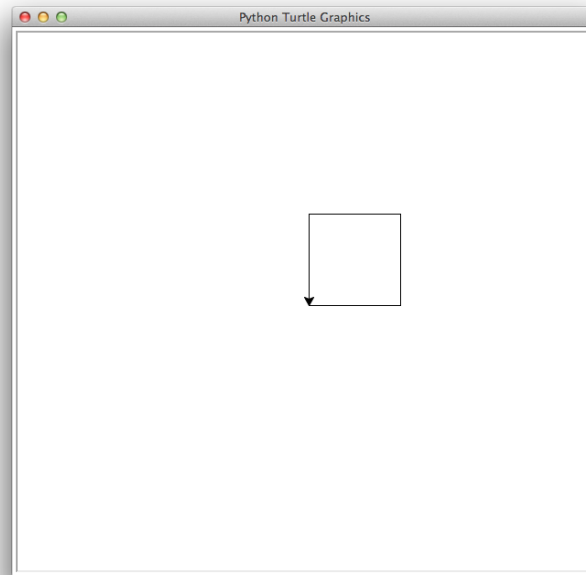
Figure 13.2: A square box.

set the absolute heading of the turtle. A heading of 0 is horizontally to the right (i.e., east), 90 is up (i.e., north), 135 is up and to the left (i.e., northwest), and so on.

Assuming you have previously entered the command of Listing 13.4, enter the commands in Listing 13.8. After doing this you should see a square drawn in the graphics window as shown in Fig. 13.2.

Listing 13.8 Commands to change the turtle's heading and draw a square box.

```
>>> t.left(90)
>>> t.fd(100)
>>> t.left(90)
>>> t.fd(100)
>>> t.left(90)
>>> t.fd(100)
```

What if we want to change the location of the turtle without generating a line? We can accomplish this by calling the method `penup()` before we enter commands to move the turtle. To re-enable drawing, we call the method `pendown()`.

We can also move the turtle to a specific position within the graphics window by using the `setposition()` method (or its aliases `setpos()` and `goto()`). `setposition()`'s arguments are the desired x and y values. The change of position does not affect the turtle's heading. If the pen is down, when you call `setposition()` (or its aliases), a straight line is drawn from that starting point to the position specified by the arguments. To demonstrate the use of `penup()`, `pendown()`, and `setposition()`, issue the commands shown in Listing 13.9. The resulting image is shown in Fig. 13.3.
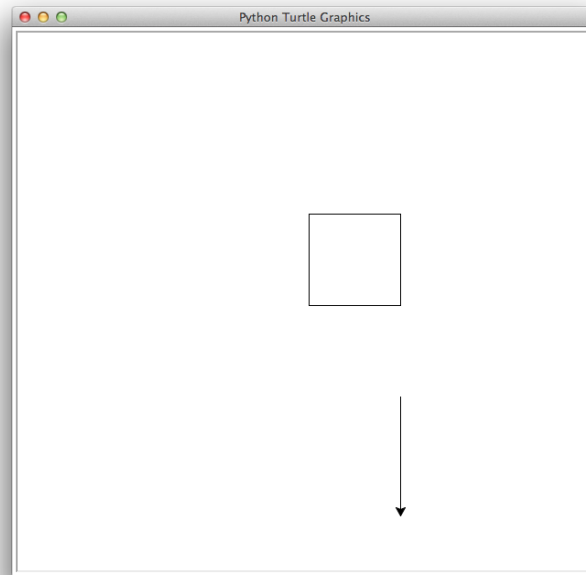
Figure 13.3: Result of moving the turtle without drawing a line and then, once at the new location, drawing a line.

Listing 13.9 Using `penup()` and `setposition()` to move the turtle without making a line.

```
>>> t.penup()
>>> t.setposition(100, -100)
>>> t.pendown()
>>> t.fd(130)
```

If we want to erase everything that we previously drew, we can use either the `clear()` or `reset()` methods. `clear()` clears the drawing from the graphics window but it leaves the turtle in its current position with its current heading. `reset()` clears the drawing and also returns the turtle to its starting position in the center of the screen. To illustrate the behavior of `clear()`, enter the statement shown in Listing 13.10.

Listing 13.10 Using the `clear()` method to clear the image.

```
>>> t.clear()
```

You should now see that the drawing that our turtle generated has been cleared from the screen but the turtle is still in the state that you last specified. To demonstrate what `reset()` does, enter the command shown in Listing 13.11.

Listing 13.11 Resetting the turtle and clearing the image.

```
>>> t.reset()
```

The turtle is moved back to the center of the screen with its original heading. Note that we do not need to call clear() before we call reset(). reset() will also clear the drawing—in the above example they were done sequentially solely for demonstrating their behavior.

13.3   Basic Shapes and Using Iteration to Generate Graphics

The commands shown in Listing 13.8 that we used to draw a square box would be rather cumbersome to type repeatedly if we wanted to draw more than one box. Observe that we are typing the same commands four times. As you already know, this sort of iteration can be accomplished in a much simpler way using a for-loop. Let's create a function called square() that draws a square using a for-loop. The function takes one argument which is the length of the square's sides. Enter the commands in Listing 13.12 that define this function and then call it three times, each time with a different length. The result should be the squares that are shown in Fig. 13.4.

Listing 13.12 A function that draws a square using a for-loop.

```
1  >>> def square(length):
2  ...        for i in range(4):
3  ...            t.fd(length)
4  ...            t.left(90)
5  ...
6  >>> square(60)
7  >>> square(100)
8  >>> square(200)
```

Building a square from straight lines is relatively straightforward, but what if we want to draw circles? Turtle provides a method called circle() that can be used to tell the turtle to draw a complete circle or only a part of a circle, i.e., an arc. The circle() method has one mandatory argument which is the radius of the circle. Optional arguments specify the "extent," which is the degrees of arc that are drawn, and the "steps," which are the number of straight-line segments used to approximate the circle. If the radius is positive, the circle is drawn (starting from the current position) by turning to the left (counterclockwise). If the radius is negative, the circle is drawn (starting from the current position) by turning to the right (clockwise). To demonstrate this, enter the commands shown in Listing 13.13. After issuing these commands, the graphics window should appear as shown in Fig. 13.5

Listing 13.13 Drawing circles using the circle() method.

```
1  >>> t.reset()        # Remove previous drawings and reset turtle.
2  >>> t.circle(100)    # Draw circle counterclockwise with radius 100.
3  >>> t.circle(-50)    # Draw circle clockwise with radius 50.
```
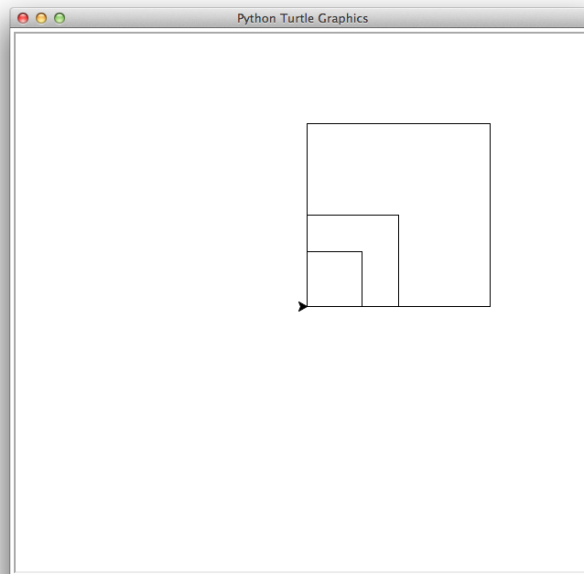
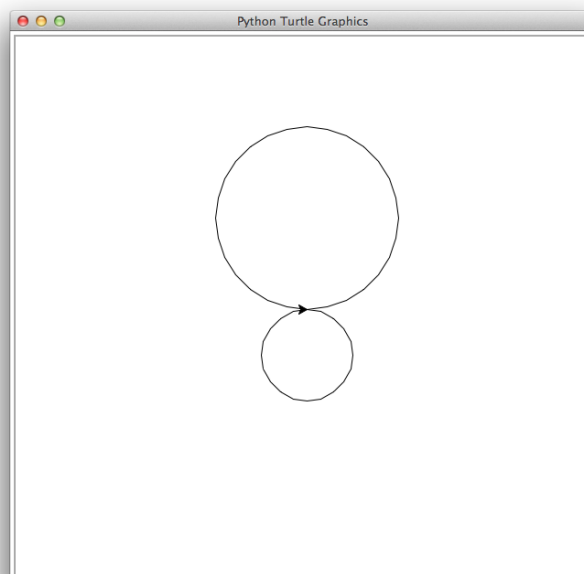Figure 13.4: Drawing multiple boxes with `square()` of Listing 13.12.

Figure 13.5: The circles drawn by the code in Listing 13.13.

13.3.1 Controlling the Turtle's Animation Speed

If you have been watching the turtle move in the graphics window as you issue commands, you will have noticed that the turtle goes through its motions relatively slowly. Sometimes watching the turtle move can be helpful but there are ways that you can speed up the movement (and hence speed up the drawing process). There is a `speed()` method for which the argument specifies the speed as an integer between 0 and 10. A value of 1 is the slowest speed. Speeds generally increase with increasing arguments so that 2 is faster than 1, three is faster than 2, and so on. However, rather than 10 being be fastest speed, it is actually the second to the fastest speed—0 is the fastest speed. The default speed is 3. To demonstrate this, issue the commands shown in Listing 13.14.

Listing 13.14 Speeding up the animation by using the `speed()` method.

```
1  >>> t.reset()
2  >>> t.speed(0)
3  >>> t.circle(100)
4  >>> t.circle(-50)
```

The image should be the same as 13.5, but it should render noticeably faster than it did previously. However, even though supplying the `speed()` method with an argument of 0 makes the animation faster, it is still quite slow if we want to draw a more complex drawing. In order to make our drawing appear almost immediately we can make use of the `tracer()` method. `tracer()` takes two arguments. One controls how often screens should be updated and the other controls the delay between these update. To obtain the fastest possible rendering, both these arguments should be set to zero as shown in Listing 13.15.

Listing 13.15 Turning off the animation with `tracer()`.

```
>>> tracer(0, 0)
```

By calling `tracer()` with bother arguments set to zero, we are essentially turning off all animation and our drawings will be drawn "immediately." However, if we turn the animation off in this way we need to explicitly update the image with the `update()` method after we are done issuing drawing commands.

If you want to reset `tracer()` to its original settings, its arguments should be 1 and 10, as shown in Listing 13.16.

Listing 13.16 Restoring animation to the default settings.

```
>>> tracer(1, 10)
```

13.4   Colors and Filled Shapes

Now that we know how to create some basic shapes, let's explore how we can create more complex images. In order to change the color of our turtle's pen we can use the `color()` method as shown in line 2 of Listing 13.17.

Listing 13.17 Changing the color of the turtle's pen.

```
1  >>> t.reset()
2  >>> t.color("blue")
```

This change the turtle's pen to blue. There are actually numerous ways of specifying a color in Python's implementation of turtle graphics. You can, as shown in Listing 13.17, specify a color via a string.[4] Alternatively, we can specify a color by providing numeric values the specify the amount of red, green, and blue. To learn more about the use of colors, use the `help()` function to read about `t.color` or `t.pencolor`.
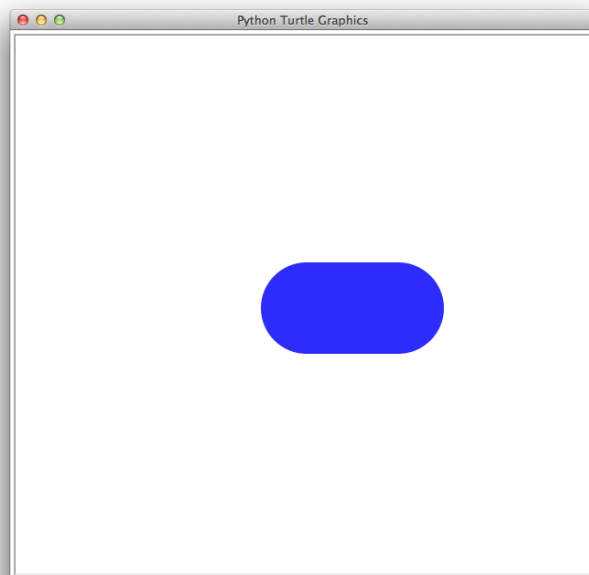

Figure 13.6: A blue line with a thickness and width of 100.

We can also change the thickness of the turtle's pen by using the `pensize()` method and passing it an integer argument that specifies the thickness. This is demonstrated in Listing 13.18. After issuing these commands (and those of Listing 13.17), you will see that a thick blue line has been drawn in the graphics window as shown in Fig. 13.6

---

[4]The string must be one of the Tk color specifications.   A listing of these can be found at www.tcl.tk/man/tcl8.4/TkCmd/colors.htm.

Listing 13.18 Changing the thickness of the turtle's pen.

```
1  >>> t.pensize(100)
2  >>> t.fd(100)
```

We can also change the background color of the graphics window. To demonstrate this, you should enter the commands of Listing 13.19.

Listing 13.19 Changing the background color of the window.

```
1  >>> t.bgcolor("blue")   # Change the background to blue.
2  >>> t.bgcolor("white")  # Change it back to white.
```

Let's take what we have learned so far and draw a more complex image. Enter the commands shown Listing 13.20. The for-loop that starts in line 3 sets the heading to angles between 0 and 345 degrees, inclusive, in 15 degree increments. For each of these headings it draws a circle with a radius of 100. (Note that a heading of 360 is the same as a heading of 0, so that why a circle is not draw with a heading of 360 degrees.) The resulting image is shown in Fig. 13.7.

Listing 13.20 Creating a red flower.

```
1  >>> t.reset()
2  >>> t.color("red")
3  >>> for angle in range(0, 360, 15):
4  ...      t.seth(angle)
5  ...      t.circle(100)
```

We can also use iteration to change the color and thickness of the turtle's pen more dynamically. As an example, try typing the commands shown in Listing 13.21. This should result in the image shown in Fig. 13.8.

Listing 13.21 Code the creation of a colorful spiral.

```
1  >>> colors = ["blue", "green", "purple", "cyan", "magenta", "violet"]
2  >>> t.reset()
3  >>> t.tracer(0, 0)
4  >>> for i in range(45):
5  ...      t.color(colors[i % 6])
6  ...      t.pendown()
7  ...      t.fd(2 + i * 5)
8  ...      t.left(45)
9  ...      t.width(i)
10 ...      t.penup()
11 ...
12 >>> t.update()
```
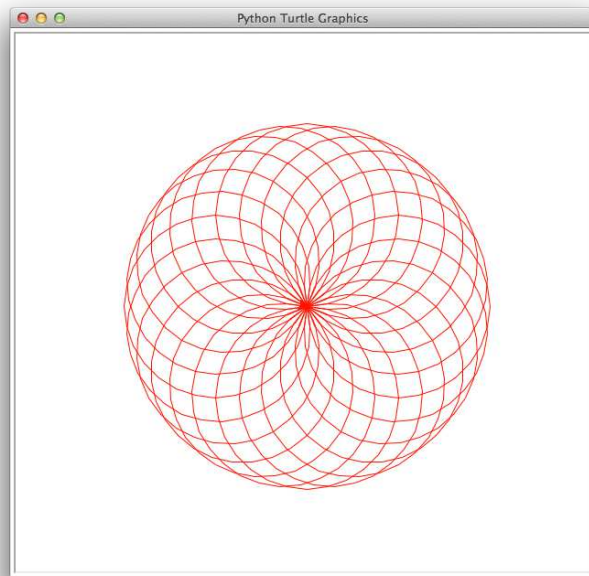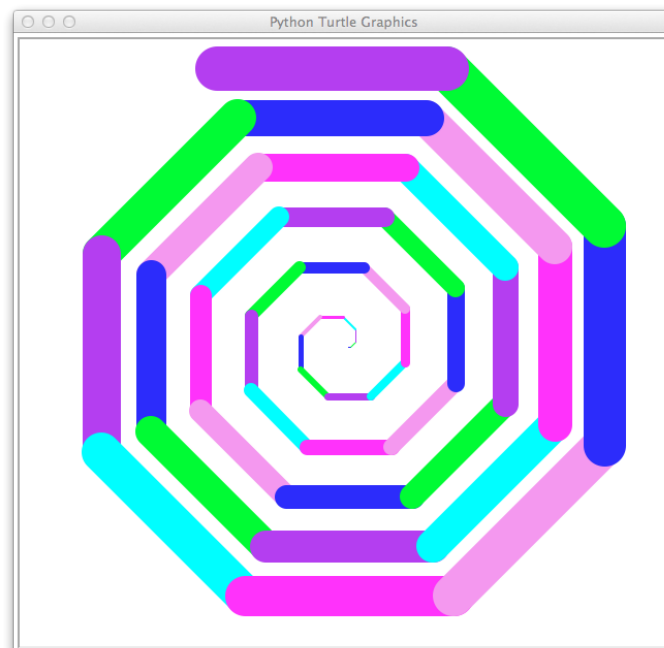
Figure 13.7: A red flower using circle().

Figure 13.8: A colorful spiral.

13.4.1 Strange Errors

As you work through the examples in this chapter you may encounter strange, complicated errors that result from small typos or bugs in your code. These errors may look alarming and confusing but you can usually discern the cause of the error if you look at the last few lines. For example, enter the command shown in line 1 of Listing 13.22.

Listing 13.22 Error example.

```
>>> t.color("pumpkin")
Traceback (most recent call last): File "<stdin>", line 1, in <module>
File "<string>", line 1, in color File
"/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/turtle.py",
line 2209, in color pcolor = self._colorstr(pcolor) File
"/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/turtle.py",
line 2689, in _colorstr return self.screen._colorstr(args) File
"/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/turtle.py",
line 1151, in _colorstr raise TurtleGraphicsError("bad color string: %s" %
str(color)) turtle.TurtleGraphicsError: bad color string: pumpkin
```

The resulting error message, shown in lines 2 through 10, is long and possibly confusing but if you look at the last line it tells you that "pumpkin" is not a valid color string.

In the introduction to this chapter we mentioned that the turtle module uses tkinter for its underlying graphics. Sometimes the errors you get will contain messages that pertain to tkinter errors by using tk in the message. If you encounter one of these errors you typically do not need to know anything about tkinter. It is likely the cause of the error is somewhere in your turtle code, e.g., you are passing a method an incorrect value, so review your code carefully and make sure it conforms to the requirements of the turtle module.

13.4.2 Filled Shapes

We can also fill the shapes we draw by using the methods begin_fill() and end_fill(). To fill a shape, we must call first begin_fill(), then issue commands to draw the desired shape, and finally call end_fill(). When we call end_fill() the shape will be filled with the currently set color. To demonstrate this, try entering the commands of Listing 13.23. After entering these commands you should see a green filled box in the graphics window as shown in Fig. 13.9.

Listing 13.23 Commands to create a filled green box.

```
>>> t.reset()
>>> t.color("green")
>>> t.begin_fill()
>>> square(100) # The square() function of Listing 13.12.
>>> t.end_fill()
```
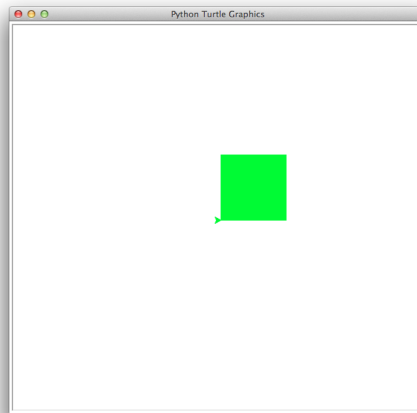
Figure 13.9: A filled green box.

You don't need to completely connect (close) the shape you want to fill. To illustrate this, try entering the code in Listing 13.24 and observe what you have drawn at each step. After issuing all these commands you should see the shape shown in Fig. 13.10

Listing 13.24 A three-sided shape that is not close.

```
1   >>> t.reset()
2   >>> t.width(10)       # Set line thickness to 10.
3   >>> t.begin_fill()    # Begin to define fill-shape.
4   >>> t.fd(150)         # Draw horizontal edge.
5   >>> t.seth(45)        # Set heading to 45 degrees.
6   >>> t.fd(150)         # Draw second edge.
7   >>> t.seth(90)        # Set heading to 90 degrees.
8   >>> t.fd(150)         # Draw vertical edge.
```

Now issue the statements shown in Listing 13.25. You should now see the image shown in Fig. 13.11. Note that the shape was filled as if there were an edge between the start-point and the end-point, but it will not draw a line along this edge. Instead, the current color is merely painted up to this edge.

Listing 13.25 Fill the shape started in Listing 13.24.

```
9    >>> t.color("blue")
10   >>> t.end_fill()
```

13.5   Visualizing Recursion

Chapter 12 provides an introduction to recursion. If you haven't done so already, it is recommended that you read that chapter before working through the examples in this section.
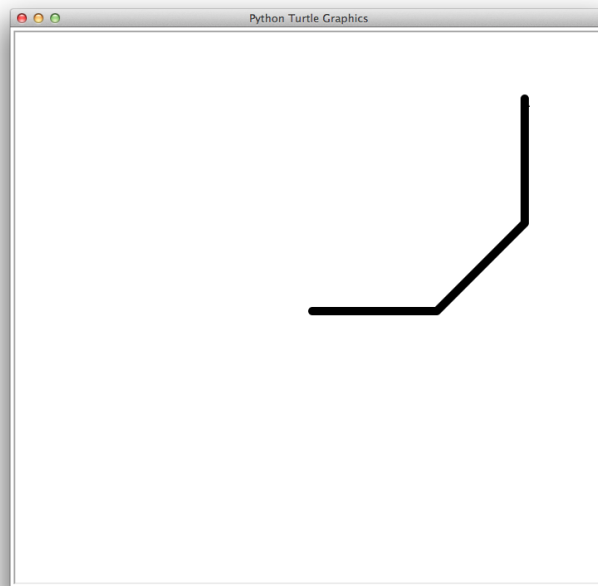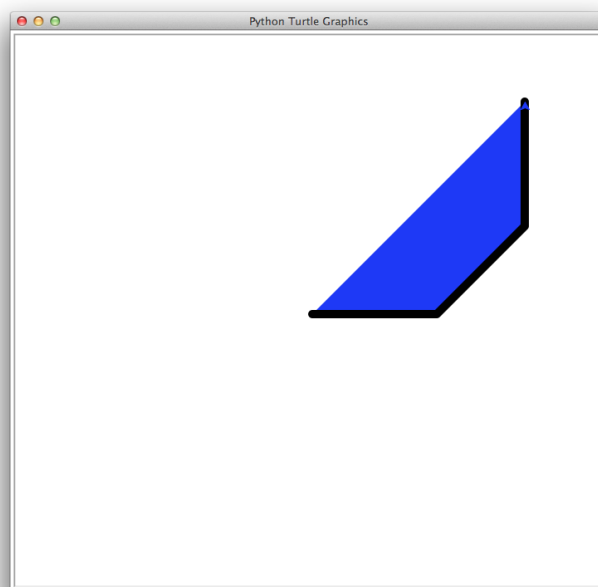
Figure 13.10: Shape that has not been closed.

Figure 13.11: A shape can be filled even if it isn't closed.

Recall the Fibonacci `fib()` function from Listing 6.23. Now consider the function `drawfib()` shown in Listing 13.26. This function takes two arguments. The second is identified as `len_ang` and, by inspection of the code, we see this parameter is used both as a length (when it is used to control movement in lines 2 and 13) and as an angle (when it used to alter the heading in lines 8, 10, and 12) The value of `len_ang` is never changed. The `drawfib()` function starts by drawing a line (line 2). Then, if the first argument is greater than 1, it alters the heading and calls itself twice, once with the value of the first argument reduced by 1 and another time with the argument reduced by 2. The final statement of the function moves the pen back to the starting point. If the first argument is 0 or 1, then the function function doesn't do anything before moving the pen back to the starting point. We accomplish doing "nothing" by using the `pass` statements given in lines 4 and 6. Note that bodies of loops, functions, conditional statements, and classes cannot be truly empty. However, we can make them effectively empty by using a `pass` statement.

Because we are not doing anything in the bodies of the first two clauses of the conditional statement, we could have eliminated lines 3 through 6 and replaced line 7 with "`if n > 1`." However, we have elected to write things as given to help establish a connection with the way numbers are generated in the Fibonacci sequence. The discussion of this code continues below the listing.

Listing 13.26 Code to draw a tree recursively. The resulting tree is related to the Fibonacci sequence.

```
1  >>> def drawfib(n, len_ang):
2  ...        t.forward(2 * len_ang)
3  ...        if n == 0:
4  ...            pass // Do nothing.
5  ...        elif n == 1:
6  ...            pass // Do nothing.
7  ...        else:
8  ...            t.left(len_ang)
9  ...            drawfib(n - 1, len_ang)
10 ...            t.right(2 * len_ang)
11 ...            drawfib(n - 2, len_ang)
12 ...            t.left(len_ang)
13 ...        t.backward(2 * len_ang)
14 ...
15 >>> # Six different starting points for six different trees.
16 >>> start_points = [[-300, 250], [-150, 250],
17 ...                  [-300, 110], [-80, 110],
18 ...                  [-300, -150], [50, -150]]
19 >>>
20 >>> # For each starting point, draw a tree with n varying
21 ... # between 1 and 6 and len_ang set to 30.
22 >>> n = 0
23 >>> for start_point in start_points:
24 ...        x, y = start_point
25 ...        n = n + 1
```

```
26  ...        t.penup()
27  ...        t.setpos(x, y)
28  ...        t.pendown()
29  ...        drawfib(n, 30)
```

It probably isn't at all obvious why we would bring up the Fibonacci sequence in connection with the code in Listing 13.26. Before reading on, you should consider entering the function and see what it produces for different arguments. Consider values of n between 1 and 10 and values of len_ang in the range of 20.

Now, to help illustrate the connection between drawfib() and the Fibonacci sequence, the code in lines x through y of Listing 13.26 draw six different trees where n varies between 1 and 6 while the len_ang is held fixed at 30. The resulting trees are shown in Fig. 13.12. The "trunk" of each tree is the horizontal line that is the right-most component of each tree. If we had drawn a tree with an n of 0, it would appear the same as with an n of 1 (i.e., the top left drawing in Fig. 13.12. If we count the number of "tips" or "branches" on the left side of the tree, for an n or 0 or 1, there is only 1 tip. For an n of 2 there are 2 (top right tree). When n is 3 there are 3 (middle left tree). When n is 4 there are 5 tips (middle right). And, we see for an n of 5 or 6, there are 8 or 13 tips, respectively (the two bottom trees). Thus, the number of "tips" in our trees corresponds to the numbers in the Fibonacci sequence! (The number of tips is the nth number in the sequence.)
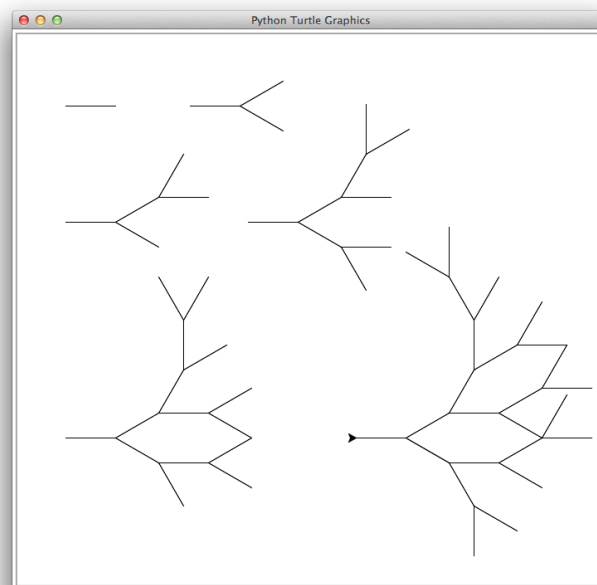

Figure 13.12: Drawings of multiple Fibonacci trees using the drawfib() function of Listing 13.26. The first argument of this function (n) varies between 1 and 6. The upper left tree was generated with an n of 1 while the bottom right tree had an n of 6.

A more complicated tree is drawn using the code shown in Listing 13.27. In this case n is 15 which results in 987 tips (although it would be almost impossible to count these from the tree figure itself). The resulting tree is shown in Fig. 13.13.
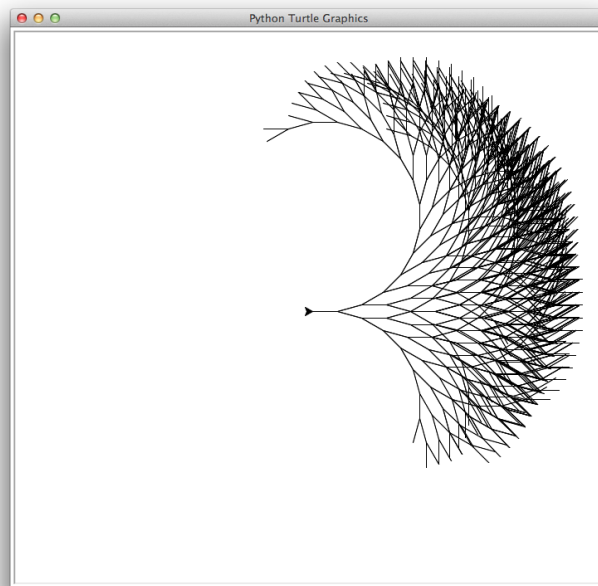
Figure 13.13: A Fibonacci tree with 987 tips.

Listing 13.27 Recursive tree using the `fib()` function with 987 branches.

```
1  >>> t.reset()
2  >>> t.tracer(0, 0)
3  >>> t.drawfib(15, 10)
4  >>> t.update()
5  >>> t.tracer(1, 10)
```

It might take a few seconds before this image appears:

Recursive definitions can also be used to draw beautiful and complex fractal images. A classic example of a fractal is a Koch snowflake. To generate a Koch snowflake, begin by entering the commands shown in Listing 13.28. The function `ks()` takes two arguments. The first is a length and the second specifies the recursive "depth" d. When d is 0, the function merely draws a straight line and returns. If d is greater than 0, the length is reduced by a factor of three and then the `ks()` function is called with this new length, with the recursion depth reduced by one-third, and with various headings. The last two lines of the listing produce the drawing shown in Fig. 13.14.

Listing 13.28 Code for producing (part of) a Koch snowflake.

```
1  >>> t.tracer(0, 0)
2  >>> def ks(length, d):
3  ...      if d == 0:
4  ...          t.forward(length)
5  ...      else:
```

```
6  ...            length = length / 3
7  ...            d = d - 1
8  ...            ks(length, d)
9  ...            t.right(60)
10 ...            ks(length, d)
11 ...            t.left(120)
12 ...            ks(length, d)
13 ...            t.right(60)
14 ...            ks(length, d)
15 ...
16 >>> ks(200, 3)
17 >>> t.update()
```
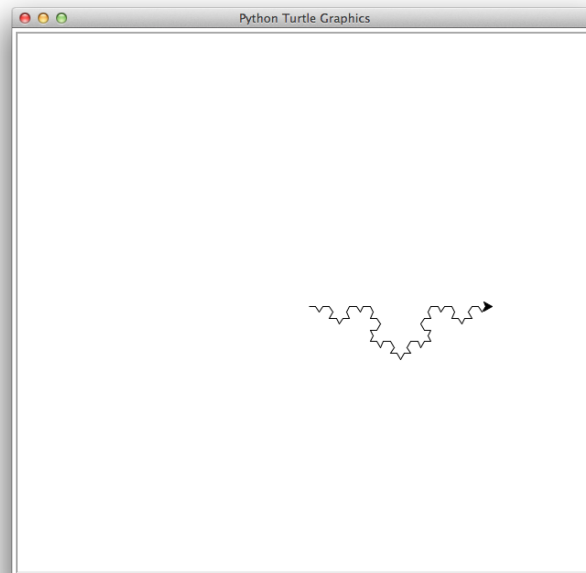

Figure 13.14: One side of a Koch snowflake.

As you can see, the ks() function only draws one side of the snowflake. If we want to complete the snowflake we can reuse this function and rotate the turtle appropriately to complete the picture. The code to accomplish this is given in Listing 13.29. The resulting drawing is shown in Fig. 13.15.

Listing 13.29 Code to produce a complete Koch snowflake.

```
1 >>> t.reset()
2 >>> colors = ["red", "orange", "pink"]
3 >>> for i in range(3):
4 ...     t.color(colors[i])
5 ...     ks(200, 3)
```

```
6  ...        t.left(120)
7  ...
8  >>> t.update()
```
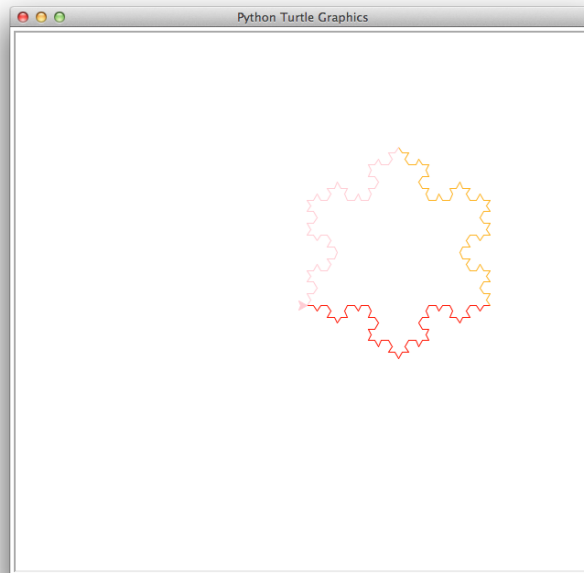

Figure 13.15: The Koch snowflake produced by the code of Listing 13.29.

13.6   Simple GUI Walk-Through

The turtle module provides simple GUI functionality. One common way GUI's interact with a user is to respond to the user's mouse clicks and perform certain actions based on those clicks. This section provides a walk-through for building a simple GUI application using "callback" functions to respond to mouse clicks. Callback functions are functions we write that specify what should be done when a particular event occurs. We need to "register" a callback function with the GUI software so that the appropriate function be called when the event occurs that the function was written to process.

13.6.1   Function References

We have previously seen that we can have a reference or alias to a `list`, i.e., we can have different identifiers that point to the same underlying memory. Also, when we pass a `list` as an actual parameter to a function, the function's formal parameter is a reference to that `list`. Python also allows us to have references to functions or, to put it another way, Python allows us to assign functions to identifiers. When we write a function name followed parentheses we are indicating that we want to call that function but what happens when we write a function name without parentheses? To find out, try typing the code shown in Listing 13.30.

Listing 13.30 Calling the `print()` function and obtaining a reference to it.

```
1  >>> print()  # This prints a newline.
2  >>> print    # This returns a reference to the print() function.
3  <built-in function print>
```

The statement in line 2 and the output in line 3 show that "calling" a function without parentheses behaves differently than when we use parentheses. In fact, without the parentheses, we are not actually calling the function. When we write `print` without parentheses we obtain a reference to the function, i.e., an object that represents the `print()` function. Thus we can assign the `print()` function to an valid identifier, as Listing 13.31 indicates.

Listing 13.31 assigning `print()` to a variable

```
1  >>> foo = print
2  >>> type(foo)
3  <class 'builtin_function_or_method'>
4  >>> foo("Hello world!")
5  Hello world!
```

The `type()` function, in line 2, shows that `foo()` is now also a function. Notice we're even told that `foo()` is a built-in function! Of course, there isn't a `foo()` function built-into Python, but at this point we have made `foo()` indistinguishable from the actual built-in function `print()`. If we try to use `foo()` in place of `print()`, we find, as lines 4 and 5 indicate, that `foo()` now behaves exactly as `print()` does!

Given that we can assign functions to identifiers, it also makes sense that we can pass functions as arguments to other functions. To demonstrate this, the `fun_run()` function in Listing 13.32 takes two arguments. The first argument is a function while the second is an argument that first argument is supposed to act upon.

Listing 13.32 Demonstration of passing a function as parameter to another function.

```
1  >>> def fun_run(func, arg):
2  ...       return func(arg)
3  ...
4  >>> fun_run(print, "Hi!")          # Apply print() to a string.
5  Hi!
6  >>> foo = print
7  >>> fun_run(foo, "Hello there!")  # Apply reference foo() to a string.
8  Hello there!
```

13.6.2   Callback functions

In order to make our GUI application respond to a user's mouse clicks, we must write a function that will be called whenever the mouse is used to click on the graphics window. First, as shown in Listing 13.33, let's define a function that takes two arguments and prints the value of those two arguments.

Listing 13.33 Function that prints its two arguments.

```
1  >>> def printxy(x, y):
2  ...        print(x, y)
```

turtle graphics has a method called `onscreenclick()` that takes a single parameter. This parameter is assumed to be a reference to a function that takes two arguments. This function will be called whenever there is a mouse click on the graphics window. The arguments passed to the function will be the x and y coordinates of the point where the click occurred. To illustrate this, assuming you have entered the code in Listing 13.33, type the code shown in Listing 13.34. In line 1 we "register" the `printxy()` function so that it will be "called back" whenever there is a click on the graphics window. If you are using IDLE, you will also need to call `mainloop()` to enable to processing of events. Once you call `mainloop()`, the interactive prompt will disappear and it won't reappear until you close the graphics window (using the close button and the top of the window).

Listing 13.34 Use of the `onscreenclick()` method to register the `printxy()` function as a callback function.

```
>>> t.onscreenclick(printxy)
>>> t.mainloop()     # If you are using IDLE.
```

Once you have done this, click on the graphics window in various places. You should see pairs of numbers corresponding to the location of the point that was clicked! Note that these pairs of numbers will be displayed in environment in which you have been issuing the Python commands—the numbers will not appear in the graphics window.

13.6.3   A simple GUI

Now let's write a more complicated turtle GUI using the `square()` function you wrote in Sec. 13.4.2. First, let's set up the window where we exercise some fine-grain control of the window's size and coordinates. The code in Listing 13.35 show how this can be accomplished using a combination of the methods `setup()`, `screensize()`, and `setworldcoordinates()`.

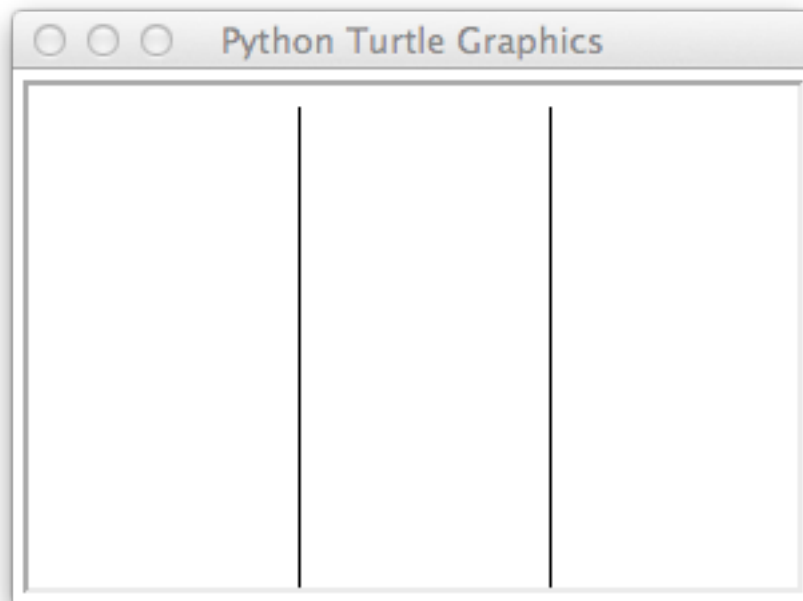**Listing 13.35** Dividing the graphics window into thirds.

Figure 13.16: Window divided into thirds.

```
1  >>> # Set window to be 300 by 200 with the point (0, 0) as the
2  >>> # lower left corner and (300, 200) as the upper right corner.
3  >>> t.setup(300, 200)
4  >>> t.screensize(300, 200)
5  >>> t.setworldcoordinates(0, 0, 300, 200)
6  >>> # Draw two vertical lines to divide the window into thirds.
7  >>> t.penup()
8  >>> t.setpos(100, 0)      # First line.
9  >>> t.pendown()
10 >>> t.setpos(100, 200)
11 >>> t.penup()
12 >>> t.setpos(200, 0)      # Second line.
13 >>> t.pendown()
14 >>> t.setpos(200, 200)
```

After issuing these commands, your window should appear as shown in Fig. 13.16.

Now let's write a callback function that will draw a shape at the point where a click occurred, but the shape that is drawn depends on where the click occurred. The function should draw a green square if the click was in the left third of the window, red circles if the click was in the middle third of the window, and blue squares if the click was in the right third of the window.

Listing 13.36 Callback function to draw various shapes at the point of a click.

```
1  >>> def shapedrawer(x, y):
2  ...       t.penup()
```

```
3  ...        # Set the position of the turtle to the clicked location.
4  ...        t.setpos(x, y)
5  ...        t.pendown()
6  ...        t.begin_fill()
7  ...        if x <= 100:          # Left third.
8  ...            t.color("green")
9  ...            square(10)
10 ...        elif 100 < x <= 200:  # Middle third.
11 ...            t.color("red")
12 ...            t.circle(10)
13 ...        else:                 # Right third.
14 ...            t.color("blue")
15 ...            square(10)
16 ...        t.end_fill()
17 ...
18 >>> t.onscreenclick(shapedrawer)
19 >>> t.mainloop()
```

The last two statements in Listing 13.36 "register" our callback function and then enter the "main loop" to start processing events. Once you have entered this code, click in various places in the graphics window. You should be able to produce images like the one shown in Fig. 13.17.
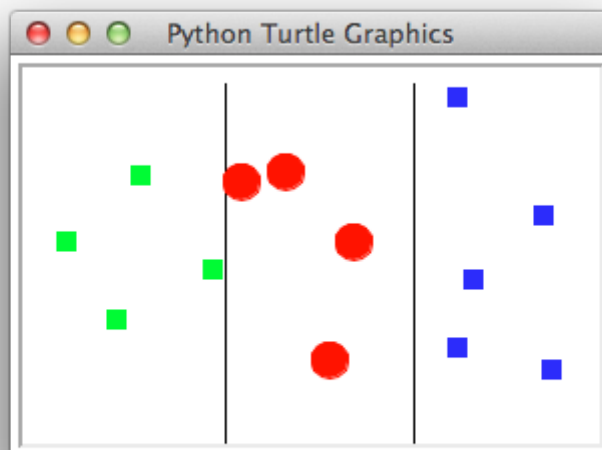

Figure 13.17: Shape GUI.