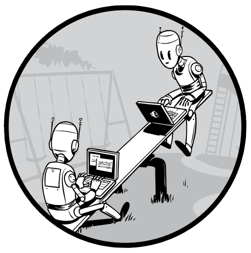
2
SPIROGRAPHS



You can use a Spirograph toy (shown in Figure 2-1) to draw mathematical curves. The toy consists of two different sized rings with plastic teeth, one large and one small. The small one has several holes. You put a pen or pencil through one of the holes and then rotate the smaller wheel inside the larger one (which has gears on its inside), keeping the pen in contact with the outer wheel, to draw an endless number of complex and wonderfully symmetric patterns.

In this project, you'll use Python to create an animation of Spirograph-like drawing curves. Our *spiro.py* program will use Python and parametric equations to describe the motion of the program's Spirograph's rings and draw the curves (which I call *spiros*). You'll save the completed drawings as PNG image files and use command line options to specify parameters or to generate random spiros.

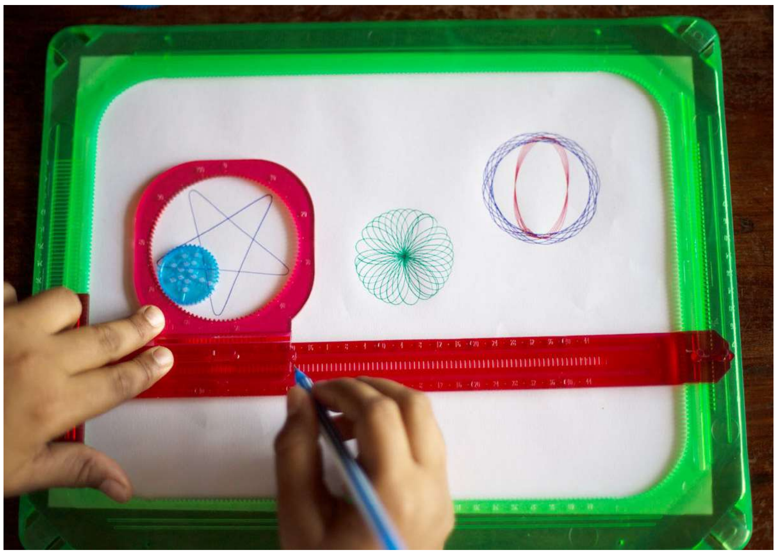


Figure 2-1: A Spirograph toy

In this project, you'll learn how to draw spiros on your computer. You'll also learn how to do the following:

- Create graphics with the `turtle` module.
- Use parametric equations.
- Use mathematical equations to generate curves.
- Draw a curve using lines.
- Use a timer to animate graphics.
- Save graphics to image files.

A word of caution about this project: I've chosen to use the `turtle` module for this project mainly for illustrative purposes and because it's fun, but `turtle` is slow and not ideal for creating graphics when performance is critical. (What do you expect from turtles?) If you want to draw something fast, there are better ways to do so, and you'll explore some of these options in upcoming projects.

Parametric Equations

In this section, you will look at a simple example of using *parametric equations* to draw a circle. Parametric equations express the coordinates of the points of a curve as functions of a variable, called a *parameter*. They make it easy to draw curves because you can just plug parameters into equations to produce a curve.

NOTE *If you'd rather not get into this math right now, you can skip ahead to the next section, which talks about the equations specific to the Spirograph project.*

Let's begin by considering that the equation used to describe a circle with radius *r*, centered at the origin of a two-dimensional plane, is. A circle consists of all the points at the x- and y-coordinates that satisfy this equation.

Now, consider the following equations:

$$x = r\cos(\theta)$$
$$y = r\sin(\theta)$$

These equations are a *parametric* representation of a circle, where the angle θ is the parameter. Any value of (x, y) in these equations will satisfy the equation for a circle described earlier, $x^2 + y^2 = r^2$. If you vary θ from 0 to 2π, you can use these equations to compute a corresponding x-and-y coordinate along the circle. Figure 2-2 shows this scheme.

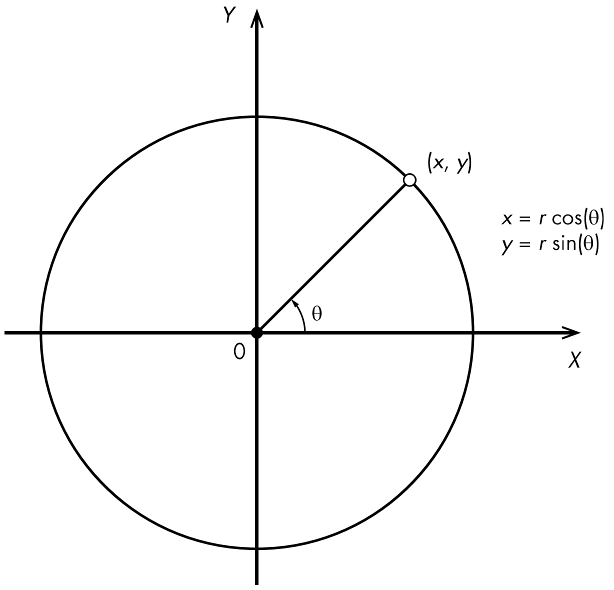


Figure 2-2: Describing a circle with a parametric equation

Remember, these two equations apply to a circle centered at the origin of the coordinate system. You can put a circle at any point in the *xy* plane by translating the center of the circle to the point (a, b). So the more general parametric equations then become $x = a + r\cos(\theta)$ and $y = b + r\cos(\theta)$. Now let's look at the equations that describe your spiros.

Spirograph Equations

Figure 2-3 shows a mathematical model of Spirograph-like motion. The model has no gears because they're used in the toy only to prevent slippage, and here you don't have to worry about anything slipping.

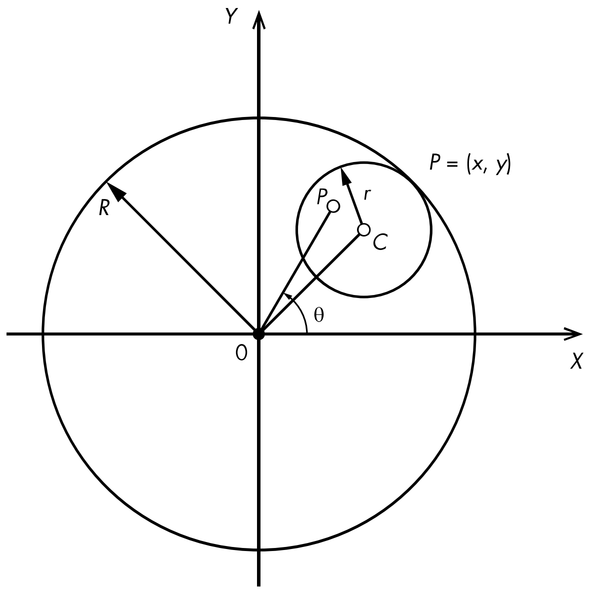


Figure 2-3: Spirograph mathematical model

In Figure 2-3, *C* is the center of the smaller circle, and *P* is the pen's tip. The radius of the bigger circle is *R*, and that of the smaller circle is *r*. You express the ratio of the radii as follows:

$$k = \frac{r}{R}$$

You express the ratio of segment *PC* to the smaller circle's radius *r* as the variable *l* (*l* = *PC* / *r*), which determines how far the pen tip is from the center of the small circle. You then combine these variables to represent the motion of *P* to produce these parametric equations:

$$x = R\left((1-k)\cos(\theta) + lk\cos\left(\frac{1-k}{k}\theta\right)\right)$$

$$y = R\left((1-k)\sin(\theta) + lk\sin\left(\frac{1-k}{k}\theta\right)\right)$$

NOTE *These curves are called hypotrochoids and epitrochoids. Although the equations may look a bit scary, the derivation is pretty straightforward. See the Wikipedia page if you'd like to explore the math.*[1]

1. *http://en.wikipedia.org/wiki/Spirograph/*

Figure 2-4 shows how you use these equations to produce a curve that varies based on the parameters used. By varying the parameters R, r, and l, you can produce an endless variety of fascinating curves.

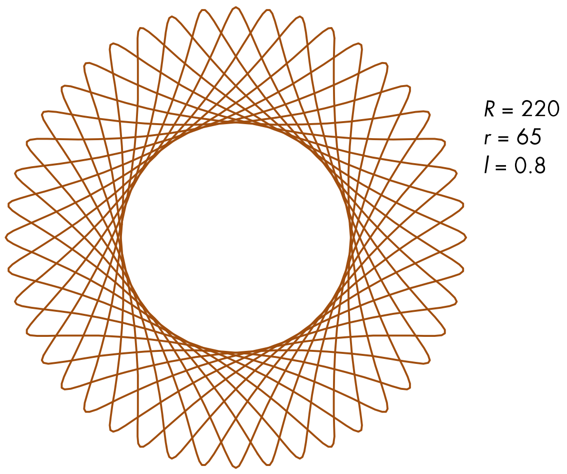


Figure 2-4: A sample curve

You draw the curve as a series of lines between points. If the points are close enough, the drawing looks like a smooth curve.

If you've played with a real Spirograph, you know that depending on the parameters used, Spirographs can require many revolutions to complete. To determine when to stop drawing, you use the *periodicity* of the Spirograph (how long before the Spirograph starts repeating itself) by looking at the ratio of the radii of the inner and outer circles:

$$\frac{r}{R}$$

You reduce this fraction by dividing the numerator and denominator by the *greatest common divisor (GCD)*, and the numerator tells you how many periods the curve needs to complete itself. For example, in Figure 2-4, the GCD of (r, R) is 5.

$$\frac{r}{R} = \frac{65}{220}$$

Here is the reduced form of this fraction:

$$\frac{(65 / 5)}{(220 / 5)} = \frac{13}{44}$$

This tells you that in 13 revolutions, the curve will start repeating itself. The number 44 tells you the number of times the smaller circle revolves about its center, which gives you a hint to the shape of the curve. If you count them in Figure 2-4, you'll see that the number of petals or lobes in the drawing is exactly 44!

Once you express the radii ratio in the reduced form r/R, the range for the parameter θ to draw the spiro is $[0, 2\pi r]$. This tells you when to stop drawing a particular spiro. Without know the ending range of the angle, you would be looping around, repeating the curve unnecessarily.

Turtle Graphics

You'll use Python's `turtle` module to create your drawings; it's a simple drawing program modeled after the idea of a turtle dragging its tail through the sand, creating patterns. The `turtle` module includes methods you can use to set the position and color of the pen (the turtle's tail) and many other useful functions for drawing. As you will see, all you need is a handful of graphics functions to create cool-looking spiros.

For example, this program uses `turtle` to draw a circle. Enter the following code, save it as *drawcircle.py*, and run it in Python:

```
  import math
❶ import turtle

  # draw the circle using turtle
  def drawCircleTurtle(x, y, r):
      # move to the start of circle
❷     turtle.up()
❸     turtle.setpos(x + r, y)
❹     turtle.down()

      # draw the circle
❺     for i in range(0, 365, 5):
❻         a = math.radians(i)
❼         turtle.setpos(x + r*math.cos(a), y + r*math.sin(a))

❽ drawCircleTurtle(100, 100, 50)
❾ turtle.mainloop()
```

You start by importing the `turtle` module at ❶. Next, you define the `drawCircleTurtle()` method, which calls `up()` at ❷. This tells Python to move the pen up; in other words, take the pen off the virtual paper so that it won't draw as you move the turtle. You want to position the turtle before you start drawing.

At ❸, you set the turtle's position to the first point on the horizontal axis: $(x + r, y)$, where (x, y) is the center of the circle. Now you're ready to draw, so you call `down()` at ❹. At ❺, you start a loop using `range(0, 365, 5)`, which increments the variable `i` in steps of 5 from 0 to 360. The `i` variable is the angle parameter you'll pass into the parametric circle equation, but first you convert it from degrees to radians at ❻. (Most computer programs require radians for angle-based calculations.)

At ❼, you compute the circle's coordinates using the parametric equations discussed previously, and set the turtle position accordingly, which draws a line from the last turtle position to the newly calculated one. (Technically, you're producing an *N*-sided polygon, but because you're using small angles, *N* will be very large, and the polygon will look like a circle.)

At ❽, you call `drawCircleTurtle()` to draw the circle, and at ❾, you call `mainloop()`, which keeps the `tkinter` window open so that you can admire your circle. (`tkinter` is the default GUI library used by Python.)

Now you're ready to draw some spiros!

Requirements

You'll use the following to create your spiros:

- The `turtle` module for drawing
- `Pillow`, a fork of the *Python Imaging Library (PIL)*, to save the spiro images

The Code

First, define a class `Spiro` to draw the curves. You'll use this class to draw a single curve in one go (using the `draw()` method) and to animate a set of random spiros using a timer and the `update()` method. To draw and animate the `Spiro` objects, you'll use a class called `SpiroAnimator`.

To see the full project code, skip ahead to "The Complete Code" on page 31.

The Spiro Constructor

Here is the `Spiro` constructor:

```
# a class that draws a Spirograph
class Spiro:
    # constructor
    def __init__(self, xc, yc, col, R, r, l):

        # create the turtle object
❶       self.t = turtle.Turtle()
        # set the cursor shape
❷       self.t.shape('turtle')
        # set the step in degrees
❸       self.step = 5
        # set the drawing complete flag
❹       self.drawingComplete = False

        # set the parameters
❺       self.setparams(xc, yc, col, R, r, l)

        # initialize the drawing
❻       self.restart()
```

The `Spiro` constructor creates a new `turtle` object at ❶, which will help you draw multiple spiros simultaneously. At ❷, you set the shape of the turtle cursor to a turtle. (You'll find other choices in the `turtle` documentation at *https://docs.python.org/3.3/library/turtle.html.*) You set the angle increment for the parametric drawing to 5 degrees at ❸, and at ❹, you set a flag that you'll use during the animation, which produces a bunch of spiros.

At ❺ and ❻, you call setup functions, as discussed next.

The Setup Functions

Let's now take a look at the `setparams()` method, which helps initialize a `Spiro` object, as shown here:

```
    # set the parameters
    def setparams(self, xc, yc, col, R, r, l):
        # the Spirograph parameters
❶       self.xc = xc
        self.yc = yc
❷       self.R = int(R)
        self.r = int(r)
        self.l = l
        self.col = col
        # reduce r/R to its smallest form by dividing with the GCD
❸       gcdVal = gcd(self.r, self.R)
❹       self.nRot = self.r//gcdVal
        # get ratio of radii
        self.k = r/float(R)
        # set the color
        self.t.color(*col)
        # store the current angle
❺       self.a = 0
```

At ❶, you store the coordinates of the center of the curve. Then you convert the radius of each circle (`R` and `r`) to an integer and store the values at ❷. At ❸, you use the `gcd()` method from the built-in Python module `fractions` to compute the GCD of the radii. You'll use this information to determine the periodicity of the curve, which you save as `self.nRot` at ❹. Finally, at ❺, you store the current angle, `a`, which you'll use to create the animation.

The restart() Method

Next, the `restart()` method resets the drawing parameters for the `Spiro` object and gets it ready for a redraw:

```
    # restart the drawing
    def restart(self):
        # set the flag
❶       self.drawingComplete = False
        # show the turtle
❷       self.t.showturtle()
```

```
        # go to the first point
❸       self.t.up()
❹       R, k, l = self.R, self.k, self.l
        a = 0.0
❺       x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
        y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
❻       self.t.setpos(self.xc + x, self.yc + y)
❼       self.t.down()
```

Here you use a Boolean flag `drawingComplete` to determine whether the drawing has been completed, and you initialize the flag at ❶. This flag is useful while multiple `Spiro` objects are being drawn because it allows you to keep track of whether a particular spiro is complete. At ❷, you show the turtle cursor, in case it was hidden. You lift up the pen at ❸ so you can move to the first position at ❻ without drawing a line. At ❹, you're just using some local variables to keep the code compact. Then, at ❺, you compute the x- and y-coordinates with the angle `a` set to 0 to get the curve's starting point. Finally, at ❼, you've finished, and you set the pen down. The `setpos()` call will draw the actual line.

The draw() Method

The `draw()` method draws the curve in one continuous line.

```
    # draw the whole thing
    def draw(self):
        # draw the rest of the points
        R, k, l = self.R, self.k, self.l
❶       for i in range(0, 360*self.nRot + 1, self.step):
            a = math.radians(i)
❷           x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
            y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
            self.t.setpos(self.xc + x, self.yc + y)
        # drawing is now done so hide the turtle cursor
❸       self.t.hideturtle()
```

At ❶, you iterate through the complete range of the parameter `i`, which is expressed in degrees as 360 times `nRot`. Compute the x- and y-coordinates for each value of the `i` parameter at ❷, and at ❸, hide the cursor because you've finished drawing.

Creating the Animation

The `update()` method shows the drawing method you use to draw the curve segment by segment to create an animation.

```
    # update by one step
    def update(self):
        # skip the rest of the steps if done
❶       if self.drawingComplete:
            return
        # increment the angle
❷       self.a += self.step
        # draw a step
        R, k, l = self.R, self.k, self.l
        # set the angle
❸       a = math.radians(self.a)
        x = self.R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
        y = self.R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
        self.t.setpos(self.xc + x, self.yc + y)
        # if drawing is complete, set the flag
❹       if self.a >= 360*self.nRot:
            self.drawingComplete = True
            # drawing is now done so hide the turtle cursor
            self.t.hideturtle()
```

At ❶, the `update()` method checks to see whether the `drawingComplete` flag is set; if not, it continues through the rest of the code. At ❷, `update()` increments the current angle. Beginning at ❸, it calculates the (x, y) position corresponding to the current angle and moves the turtle there, drawing the line segment in the process.

When I discussed the Spirograph equations, I talked about the periodicity of the curve. A Spirograph starts repeating itself after a certain angle. At ❹, you see whether the angle has reached the full range computed for this particular curve. If so, you set the `drawingComplete` flag because the drawing is complete. Finally, you hide the turtle cursor so you can see your beautiful creation.

The SpiroAnimator Class

The `SpiroAnimator` class will let you draw random spiros simultaneously. This class uses a timer to draw the curves one segment at a time; this technique updates the graphics periodically and lets the program process events such as button presses, mouse clicks, and so on. But this timer technique requires some restructuring in the drawing code.

```
# a class for animating Spirographs
class SpiroAnimator:
    # constructor
    def __init__(self, N):
        # set the timer value in milliseconds
❶       self.deltaT = 10
        # get the window dimensions
❷       self.width = turtle.window_width()
        self.height = turtle.window_height()
        # create the Spiro objects
❸       self.spiros = []
        for i in range(N):
            # generate random parameters
❹           rparams = self.genRandomParams()
            # set the spiro parameters
❺           spiro = Spiro(*rparams)
            self.spiros.append(spiro)
        # call timer
❻       turtle.ontimer(self.update, self.deltaT)
```

At ❶, the `SpiroAnimator` constructor sets `deltaT` to 10, which is the time interval in milliseconds you'll use for the timer. At ❷, you store the dimensions of the turtle window. Then you create an empty array at ❸, which you'll populate with `Spiro` objects. These encapsulate the Spirograph drawing and then loop *N* times (`N` is passed into `SpiroAnimator` in the constructor), create a new `Spiro` object at ❺, and add it to the list of `Spiro` objects. The `rparams` here is a tuple that you need to pass into the `Spiro` constructor. However, the constructor expects a list of arguments, so you use the Python `*` operator to convert a tuple to a list of arguments.

Finally, at ❻, you set the `turtle.ontimer()` method to call `update()` every `deltaT` milliseconds.

Notice at ❹ that you call a helper method called `genRandomParams()`. You'll look at that next.

The genRandomParams() Method

You'll use the `genRandomParams()` method to generate random parameters to send to each `Spiro` object as it's created in order to create a wide variety of curves.

```
    # generate random parameters
    def genRandomParams(self):
        width, height = self.width, self.height
❶       R = random.randint(50, min(width, height)//2)
❷       r = random.randint(10, 9*R//10)
❸       l = random.uniform(0.1, 0.9)
❹       xc = random.randint(-width//2, width//2)
❺       yc = random.randint(-height//2, height//2)
❻       col = (random.random(),
               random.random(),
               random.random())
❼       return (xc, yc, col, R, r, l)
```

To generate random numbers, you use two methods from the `random` Python module: `randint()`, which returns random integers in the specified range, and `uniform()`, which does the same for floating-point numbers. At ❶, you set `R` to a random integer between 50 and the value of half the smallest dimension of your window, and at ❷, you set `r` to between 10 and 90 percent of `R`.

Then at ❸, you set *l* to a random fraction between 0.1 and 0.9. At ❹ and ❺, you select a random point on the screen to place the center of the spiro by selecting random x- and y-coordinates from within the screen boundaries. Assign a random color to the curve at ❻ by setting random values to the red, green, and blue color components. Finally, at ❼, all of your calculated parameters are returned as a tuple.

Restarting the Program

We'll use another `restart()` method to restart the program.

```
# restart spiro drawing
    def restart(self):
        for spiro in self.spiros:
            # clear
            spiro.clear()
            # generate random parameters
            rparams = self.genRandomParams()
            # set the spiro parameters
            spiro.setparams(*rparams)
            # restart drawing
            spiro.restart()
```

This loops through all the `Spiro` objects, clears the previous drawing for each, assigns new spiro parameters, and then restarts the program.

The update() Method

The following code shows the `update()` method in `SpiroAnimator`, which is called by the timer to update all the `Spiro` objects used in the animation:

```
      def update(self):
          # update all spiros
❶         nComplete = 0
          for spiro in self.spiros:
              # update
❷             spiro.update()
              # count completed spiros
❸             if spiro.drawingComplete:
                  nComplete += 1
          # restart if all spiros are complete
❹         if nComplete == len(self.spiros):
              self.restart()
          # call the timer
❺         turtle.ontimer(self.update, self.deltaT)
```

The `update()` method uses a counter `nComplete` to track the number of `Spiro` objects being drawn. After you initialize at ❶, it loops through the list of `Spiro` objects, updates them at ❷, and increments the counter at ❸ if a `Spiro` is completed.

Outside the loop at ❹, you check the counter to determine whether all the objects have finished drawing. If so, you restart the animation with fresh spiros by calling the `restart()` method. At the end of a `restart()` at ❺, you call the timer method, which calls `update()` again after `deltaT` milliseconds.

Showing or Hiding the Cursor

Finally, you use the following method to toggle the turtle cursor on and off. This can be used to make the drawing go faster.

```
    # toggle turtle cursor on and off
    def toggleTurtles(self):
        for spiro in self.spiros:
            if spiro.t.isvisible():
                spiro.t.hideturtle()
            else:
                spiro.t.showturtle()
```

Saving the Curves

Use the `saveDrawing()` method to save the drawings as PNG image files.

```
  # save drawings as PNG files
  def saveDrawing():
      # hide the turtle cursor
❶         turtle.hideturtle()
          # generate unique filenames
❷         dateStr = (datetime.now()).strftime("%d%b%Y-%H%M%S")
          fileName = 'spiro-' + dateStr
          print('saving drawing to %s.eps/png' % fileName)
          # get the tkinter canvas
❸         canvas = turtle.getcanvas()
          # save the drawing as a postscipt image
❹         canvas.postscript(file = fileName + '.eps')
          # use the Pillow module to convert the postscript image file to PNG
❺         img = Image.open(fileName + '.eps')
❻         img.save(fileName + '.png', 'png')
          # show the turtle cursor
❼         turtle.showturtle()
```

At ❶, you hide the turtle cursor so that you won't see it in the final drawing. Then, at ❷, you use `datetime()` to generate unique names for the image files by using the current time and date (in the *day-month-year-hour-minute-second* format). You append this string to *spiro-* to generate the filename.

The `turtle` program uses user interface (UI) windows created by `tkinter`, and you use the `canvas` object of `tkinter` to save the window in the Embedded PostScript (EPS) file format at ❸ and ❹. Because EPS is vector based, you can use it to print your images at high resolution, but PNG is more versatile, so you use `Pillow` to open the EPS file at ❺ and save it as a PNG file at ❻. Finally, at ❼, you unhide the turtle cursor.

Parsing Command Line Arguments and Initialization

Like in Chapter 1, you use `argparse` in the `main()` method to parse command line options sent to the program.

```
❶   parser = argparse.ArgumentParser(description=descStr)

    # add expected arguments
❷   parser.add_argument('--sparams', nargs=3, dest='sparams', required=False,
                        help="The three arguments in sparams: R, r, l.")

    # parse args
❸   args = parser.parse_args()
```

At ❶, you create the argument parser object, and at ❷, you add the `--sparams` optional argument to the parser. You make the call that does the actual parsing at ❸.

Next, the code sets up some `turtle` parameters.

```
    # set the width of the drawing window to 80 percent of the screen width
❶   turtle.setup(width=0.8)

    # set the cursor shape to turtle
❷   turtle.shape('turtle')

    # set the title to Spirographs!
❸   turtle.title("Spirographs!")
    # add the key handler to save our drawings
❹   turtle.onkey(saveDrawing, "s")
    # start listening
❺   turtle.listen()

    # hide the main turtle cursor
❻   turtle.hideturtle()
```

At ❶, you use `setup()` to set the width of the drawing window to 80 percent of the screen width. (You could also give `setup()` specific height and origin parameters.) You set the cursor shape to `turtle` at ❷, and you set the title of the program window to *Spirographs!* at ❸. At ❹, you use `onkey()` with `saveDrawing` to save the drawing when you press S. Then, at ❺, you call `listen()` to make the window listen for user events. Finally, at ❻, you hide the turtle cursor.

Once the command line arguments are parsed, the rest of the code proceeds as follows:

```
    # check for any arguments sent to --sparams and draw the Spirograph
❶   if args.sparams:
❷       params = [float(x) for x in args.sparams]
        # draw the Spirograph with the given parameters
        col = (0.0, 0.0, 0.0)
❸       spiro = Spiro(0, 0, col, *params)
❹       spiro.draw()
```

```
    else:
        # create the animator object
❺       spiroAnim = SpiroAnimator(4)
        # add a key handler to toggle the turtle cursor
❻       turtle.onkey(spiroAnim.toggleTurtles, "t")
        # add a key handler to restart the animation
❼       turtle.onkey(spiroAnim.restart, "space")

    # start the turtle main loop
❽   turtle.mainloop()
```

At ❶, you first check whether any arguments were given to `--sparams`; if so, you extract them from the string and use a *list comprehension* to convert them into floats at ❷. (A list comprehension is a Python construct that lets you create a list in a compact and powerful way. For example, `a = [2*x for x in range(1, 5)]` creates a list of the first four even numbers.)

At ❸, you use any extracted parameters to construct the `Spiro` object (with the help of the Python `*` operator, which converts the list into arguments). Then, at ❹, you call `draw()`, which draws the spiro.

Now, if no arguments were specified on the command line, you enter random mode. At ❺, you create a `SpiroAnimator` object, passing it the argument `4`, which tells it to create four drawings. At ❻, use `onkey` to capture any presses of the T key so that you can use it to toggle the turtle cursors (`toggleTurtles`), and at ❼, handle presses of the spacebar (`space`) so that you can use it to restart the animation at any point. Finally, at ❽, you call `mainloop()` to tell the `tkinter` window to stay open, listening for events.

The Complete Code

Here is the complete Spirograph program. You can also download the code for this project from *https://github.com/electronut/pp/blob/master/spirograph/spiro.py*.

```
import sys, random, argparse
import numpy as np
import math
import turtle
import random
from PIL import Image
from datetime import datetime
from fractions import gcd

# a class that draws a Spirograph
class Spiro:
    # constructor
    def __init__(self, xc, yc, col, R, r, l):

        # create the turtle object
        self.t = turtle.Turtle()
        # set the cursor shape
        self.t.shape('turtle')
        # set the step in degrees
        self.step = 5
        # set the drawing complete flag
        self.drawingComplete = False

        # set the parameters
        self.setparams(xc, yc, col, R, r, l)

        # initialize the drawing
        self.restart()

    # set the parameters
    def setparams(self, xc, yc, col, R, r, l):
        # the Spirograph parameters
        self.xc = xc
        self.yc = yc
        self.R = int(R)
        self.r = int(r)
        self.l = l
        self.col = col
        # reduce r/R to its smallest form by dividing with the GCD
        gcdVal = gcd(self.r, self.R)
        self.nRot = self.r//gcdVal
        # get ratio of radii
        self.k = r/float(R)
        # set the color
        self.t.color(*col)
        # store the current angle
        self.a = 0

    # restart the drawing
    def restart(self):
        # set the flag
        self.drawingComplete = False
        # show the turtle
        self.t.showturtle()
        # go to the first point
        self.t.up()
        R, k, l = self.R, self.k, self.l
        a = 0.0
        x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
        y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
        self.t.setpos(self.xc + x, self.yc + y)
        self.t.down()

    # draw the whole thing
    def draw(self):
        # draw the rest of the points
        R, k, l = self.R, self.k, self.l
        for i in range(0, 360*self.nRot + 1, self.step):
            a = math.radians(i)
            x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
            y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
            self.t.setpos(self.xc + x, self.yc + y)
            # drawing is now done so hide the turtle cursor
            self.t.hideturtle()

    # update by one step
    def update(self):
        # skip the rest of the steps if done
        if self.drawingComplete:
            return
        # increment the angle
        self.a += self.step
        # draw a step
        R, k, l = self.R, self.k, self.l
        # set the angle
        a = math.radians(self.a)
        x = self.R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
        y = self.R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
        self.t.setpos(self.xc + x, self.yc + y)
        # if drawing is complete, set the flag
        if self.a >= 360*self.nRot:
            self.drawingComplete = True
            # drawing is now done so hide the turtle cursor
            self.t.hideturtle()

    # clear everything
    def clear(self):
        self.t.clear()

# a class for animating Spirographs
class SpiroAnimator:
    # constructor
    def __init__(self, N):
        # set the timer value in milliseconds
        self.deltaT = 10
        # get the window dimensions
        self.width = turtle.window_width()
        self.height = turtle.window_height()
        # create the Spiro objects
        self.spiros = []
        for i in range(N):
            # generate random parameters
            rparams = self.genRandomParams()
            # set the spiro parameters
            spiro = Spiro(*rparams)
            self.spiros.append(spiro)
        # call timer
        turtle.ontimer(self.update, self.deltaT)

    # restart spiro drawing
    def restart(self):
        for spiro in self.spiros:
            # clear
            spiro.clear()
            # generate random parameters
            rparams = self.genRandomParams()
```

```
                # set the spiro parameters
                spiro.setparams(*rparams)
                # restart drawing
                spiro.restart()

        # generate random parameters
        def genRandomParams(self):
            width, height = self.width, self.height
            R = random.randint(50, min(width, height)//2)
            r = random.randint(10, 9*R//10)
            l = random.uniform(0.1, 0.9)
            xc = random.randint(-width//2, width//2)
            yc = random.randint(-height//2, height//2)
            col = (random.random(),
                   random.random(),
                   random.random())
            return (xc, yc, col, R, r, l)

        def update(self):
            # update all spiros
            nComplete = 0
            for spiro in self.spiros:
                # update
                spiro.update()
                # count completed spiros
                if spiro.drawingComplete:
                    nComplete += 1
            # restart if all spiros are complete
            if nComplete == len(self.spiros):
                self.restart()
            # call the timer
            turtle.ontimer(self.update, self.deltaT)

        # toggle turtle cursor on and off
        def toggleTurtles(self):
            for spiro in self.spiros:
                if spiro.t.isvisible():
                    spiro.t.hideturtle()
                else:
                    spiro.t.showturtle()

    # save drawings as PNG files
    def saveDrawing():
        # hide the turtle cursor
        turtle.hideturtle()
        # generate unique filenames
        dateStr = (datetime.now()).strftime("%d%b%Y-%H%M%S")
        fileName = 'spiro-' + dateStr
        print('saving drawing to %s.eps/png' % fileName)
        # get the tkinter canvas
        canvas = turtle.getcanvas()
        # save the drawing as a postscipt image
        canvas.postscript(file = fileName + '.eps')
        # use the Pillow module to convert the poscript image file to PNG
        img = Image.open(fileName + '.eps')
        img.save(fileName + '.png', 'png')
        # show the turtle cursor
        turtle.showturtle()

# main() function
def main():
    # use sys.argv if needed
    print('generating spirograph...')
    # create parser
    descStr = """This program draws Spirographs using the Turtle module.
    When run with no arguments, this program draws random Spirographs.

    Terminology:

    R: radius of outer circle
    r: radius of inner circle
    l: ratio of hole distance to r
    """
    parser = argparse.ArgumentParser(description=descStr)

    # add expected arguments
    parser.add_argument('--sparams', nargs=3, dest='sparams', required=False,
                        help="The three arguments in sparams: R, r, l.")

    # parse args
    args = parser.parse_args()

    # set the width of the drawing window to 80 percent of the screen width
    turtle.setup(width=0.8)

    # set the cursor shape to turtle
    turtle.shape('turtle')

    # set the title to Spirographs!
    turtle.title("Spirographs!")
    # add the key handler to save our drawings
    turtle.onkey(saveDrawing, "s")
    # start listening
    turtle.listen()

    # hide the main turtle cursor
    turtle.hideturtle()

    # check for any arguments sent to --sparams and draw the Spirograph
    if args.sparams:
        params = [float(x) for x in args.sparams]
        # draw the Spirograph with the given parameters
        col = (0.0, 0.0, 0.0)
        spiro = Spiro(0, 0, col, *params)
        spiro.draw()
    else:
        # create the animator object
        spiroAnim = SpiroAnimator(4)
        # add a key handler to toggle the turtle cursor
        turtle.onkey(spiroAnim.toggleTurtles, "t")
        # add a key handler to restart the animation
        turtle.onkey(spiroAnim.restart, "space")

    # start the turtle main loop
    turtle.mainloop()

# call main
if __name__ == '__main__':
    main()
```

Running the Spirograph Animation

Now it's time to run your program.

```
$ python spiro.py
```

By default, the *spiro.py* program draws random spiros, as shown in Figure 2-5. Pressing S saves the drawing.

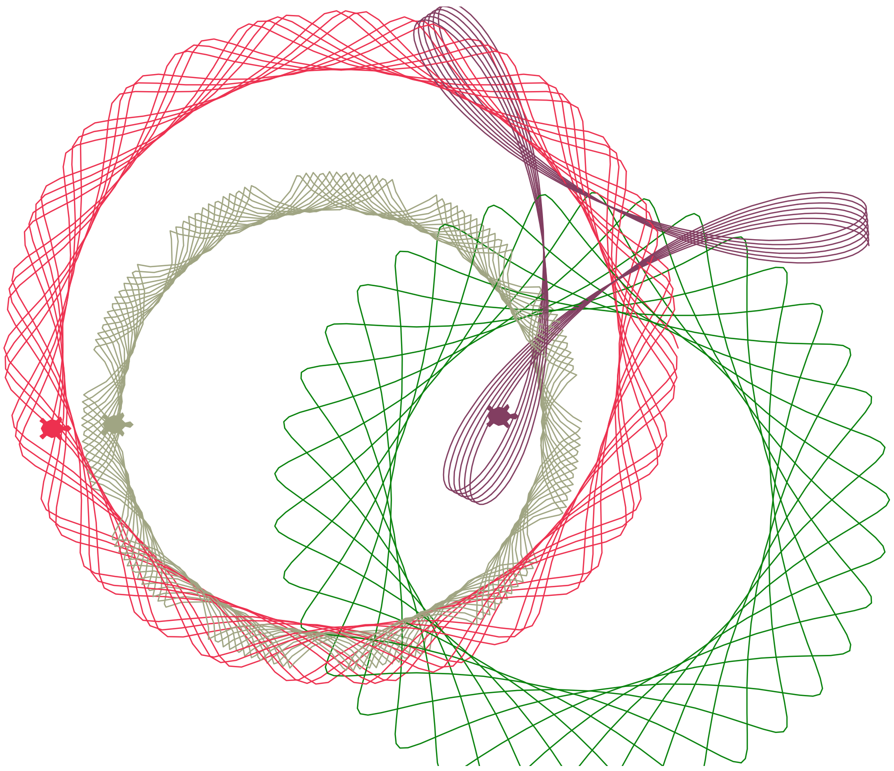


Figure 2-5: A sample run of spiro.py

Now run the program again, this time passing in parameters on the command line to draw a particular spiro.

```
$ python spiro.py --sparams 300 100 0.9
```

Figure 2-6 shows the output. As you can see, this code draws a single spiro with the parameters specified by the user, in contrast to Figure 2-5, which displays an animation of several random spiros.

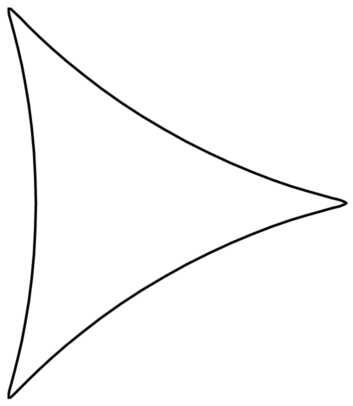


Figure 2-6: A sample run of spiro.py *with specific parameters*

Summary

In this project, you learned how to create Spirograph-like curves. You also learned how to adjust the input parameters to generate a variety of different curves and to animate them on screen. I hope you enjoy creating these spiros. (You'll find a surprise in Chapter 13, where you'll learn how to project spiros onto a wall!)

Experiments!

Here are some ways to experiment further with spiros.

1. Now that you know how to draw circles, write a program to draw random *spirals*. Find the equation for a *logarithmic spiral* in parametric form and then use it to draw the spirals.
2. You might have noticed that the turtle cursor is always oriented to the right as the curves are drawn, but that's not how turtles move! Orient the turtle so that, as the curve is being drawn, it faces in the direction of drawing. (Hint: calculate the direction vector between successive points for every step and reorient the turtle using the `turtle.setheading()` method.)

3. Try drawing a Koch snowflake, a fractal curve constructed using recursion (a function that calls itself), with the turtle. You can structure your recursive function call like this:

```
# recursive Koch snowflake
def kochSF(x1, y1, x2, y2, t):
    # compute intermediate points p2, p3
    if segment_length > 10:
        # recursively generate child segments
        # flake #1
        kochSF(x1, y1, p1[0], p1[1], t)
        # flake #2
        kochSF(p1[0], p1[1], p2[0], p2[1], t)
        # flake #3
        kochSF(p2[0], p2[1], p3[0], p3[1], t)
        # flake #4
        kochSF(p3[0], p3[1], x2, y2, t)
    else:
        # draw
        # ...
```

If you get really stuck, you can find my solution at *http://electronut.in/koch-snowflake-and-the-thue-morse-sequence/*.