600.112: Intro Programming for Scientists and Engineers

Assignment 1: Turtle Graphics

Peter H. Fröhlich
phf@cs.jhu.edu

Joanne Selinski
joanne@cs.jhu.edu

Due Date: Wednesdays 9/16 and 9/23

Introduction

The first actual programming assignment for *600.112: Introductory Programming for Scientists and Engineers* is meant to really get you going with programming in Python. However, it does not yet cover any specific application area in science or engineering, it is *only* about programming. We will focus on drawing things on the screen using Python's Turtle Graphics module to make this journey more entertaining.

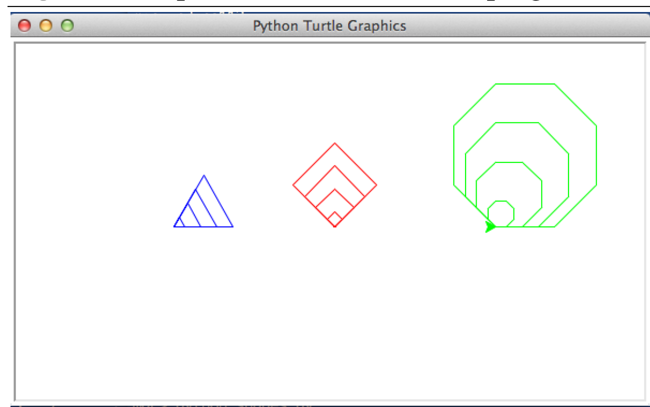
There are three things to do: First you'll write a program that draws a number of basic geometric shapes on the screen. Second you'll write a program that will plot one period of the cosine function. Third you'll write a program that will plot a parametric curve where the x and y positions are derived from a single parameter in an interesting way.

There are detailed submission instructions on Blackboard which you should follow **to the letter**! You can **lose points** if you create more work than necessary for the graders by not following the instructions.

1 Geometric Shapes [5 points]

The first program you will write draws a number of simple geometric shapes: triangles, diamonds, and octagons (the eight-sided regular polygon). Please call your program `shapes.py` and nothing else! Figure 1 shows what the output of your program will look like.

Figure 1 Output of the `shapes.py` program.



For this and the following programs, you will need to use *functions* and *for loops* as discussed in lecture. This is in addition to your basic understanding of expressions and variables. We will, however, lead you through the problems rather slowly and with a lot of advice on how to proceed, so you should be okay as long as you follow along diligently.

Before you can do anything else, you'll need a very basic first version of your program that sets up the `turtle` module and properly waits for the user to close the window. Here is what that first version could look like:

```
import turtle

def main():
    turtle.setup()
    turtle.done()

main()
```

This version will literally do nothing but open the turtle window, wait for the user to close the window, and exit. Once you have this, you can write your first function, probably the one to draw triangles. As you can see from Figure 1, you will have to draw triangles of different sizes, so your function should take a `size` parameter. After decid-

ing these two things, you can add the function and some code to test it to your program:

```
import turtle

def triangle(size):
    pass

def main():
    turtle.setup()
    triangle(100)
    turtle.done()

main()
```

If you run this version, it will behave just like the first one; however, you can now be sure that you didn't make a mistake in defining the function and calling it. The next step is to develop the body of the `triangle` function. Drawing a triangle requires that we move forward three times and turn left three times. We should move forward each time by `size`; we need to turn left each time by 120 degrees (why?). So one step in the process of drawing a triangle is to (a) move forward and (b) turn; we need to perform this step three times, so we put it inside a `for` loop:

```
import turtle

def triangle(size):
    for _ in range(3):
        turtle.forward(size)
        turtle.left(120)

def main():
    turtle.setup()
    triangle(100)
    turtle.done()

main()
```

When you run this version of the program, you should see a single triangle on the screen. While that's a far cry from the final image you're supposed to draw, it's certainly progress!

The process we just illustrated, starting with a very simple program and testing it, adding a little bit of code and testing again, adding a little more code and testing again, etc. is **very** important. It's called **iterative development**, and we will be emphasizing this technique throughout the course! If you sit down and write code for two hours before ever testing your program, you will be overwhelmed by all the things that are going wrong. If instead you write code for only two minutes and test your program again, there is much less code that can go wrong, and therefore it will be a lot easier for you to correct your mistakes. **Always program in baby steps!**

Now that you have a working `triangle` function, you can write and test a `diamond` function a similar manner. Your `diamond` function will also have a single parameter which is the length of each side. Each diamond will have 4 sides, and the angles between them should be 90 degrees each. Essentially our diamond is a square rotated so that the point is at the top and bottom.

First you may want to write the code to draw a square. This will be very similar to the triangle function. However, in order to turn it into a diamond, you must turn the turtle 45 degrees to the left before it starts to draw the shape.

We'll leave the function empty in the following so you have something to experiment with on your own, especially since it's very similar to `triangle` anyway. Remember to write it in baby steps and test as you go along.

This brings us to the third shape you need to draw, the octagon. We could write a function that draws **just** an octagon, but you've probably noticed by now that the main difference between all regular polygons is (a) how many lines to draw and (b) how much to turn left between each line. So instead of writing an `octagon` function, let's write a `polygon` function that can draw **any** regular polygon we desire. This is an example of **abstraction**, generalizing a problem and developing a solution to an entire class of problems instead of one specific instance. It's a feature of programming that we should take advantage of and learn to do well.

Obviously it's not enough to tell the `polygon` function how big of a polygon to draw, we also have to tell it how many sides the polygon is supposed to have:

```
import turtle

def triangle(size):
    for _ in range(3):
        turtle.forward(size)
        turtle.left(120)

...

def polygon(size, sides):
    pass

def main():
    turtle.setup()
    triangle(100)
```

```
    diamond(100)
    polygon(100, 8)
    turtle.done()

main()
```

Looking back at the structure of our previous functions, it should be clear that the `for` loop in `polygon` has to run `sides` times: it ran three times for a triangle, should run four times for a square or diamond, so it has to run eight times for an octagon, five times for a pentagon, and so on. The angle by which we turn after each line has to depend on the number of sides as well: If we add up the angles for the triangle and the rectangle, we get 360 each time; so for an arbitrary polygon, the angle should be $360/n$ where n is the number of sides. We can put the expression to do this calculation directly into the function call like this:

```
def polygon(size, sides):
    for _ in range(sides):
        turtle.forward(size)
        turtle.left(360.0/sides)
```

However, that causes it to be recalculated each time that statement is executed by the loop. Instead, we can give that value a name such as `angle` before the loop starts, and then refer to it as such in the function call. So here we go:

```
import turtle

def triangle(size):
    for _ in range(3):
        turtle.forward(size)
        turtle.left(120)


...

def polygon(size, sides):
    angle = 360.0 / sides
    for _ in range(sides):
        turtle.forward(size)
        turtle.left(angle)

def main():
    turtle.setup()
    triangle(100)
    diamond(100)
    polygon(100, 8)
    turtle.done()

main()
```

Note that it is **very** important that we write `360.0` when calculating the angle and not just `360` (why?).

At this point we can draw all the required shapes, so what we have left to do is draw them multiple times, in different colors, and at different positions. Colors are the easiest thing to get a handle on, so let's start there. The `turtle` module provides a `color` function that we can use: we simply say `turtle.color("red")` for example. Figure 1 indicates that we need green, blue, and red for triangles, diamonds, and octagons respectively, so we change our code as follows:

```
import turtle

...

def main():
    turtle.setup()
    turtle.color("blue")
    triangle(100)
    turtle.color("red")
    diamond(100)
    turtle.color("green")
    polygon(100, 8)
    turtle.done()

main()
```

Drawing each shape repeatedly at different sizes is obviously something a `for` loop can do. Each shape already takes a size parameter, so all we need to know is what sizes we are supposed to draw them at. Let's say we want to draw each shape at a size of 10, 25, 40, and 55, so each shape gets drawn four times. Creating a `for` loop that goes through these values requires that we use the three-argument form of the `range` function: `range(10, 56, 15)` Remember that the lower bound is inclusive while the upper bound is exclusive; if we would use `55` instead of `56`, the value 55 itself wouldn't be included. Let's write a separate function for drawing our set of four triangles as follows:

```
import turtle

...

def triangles():
    for size in range(10, 56, 15):
        triangle(size)

def main():
    turtle.setup()
    turtle.color("blue")
    triangles()
    turtle.color("red")
    rectangle(100)
```

```
    turtle.color("green")
    polygon(100, 8)
    turtle.done()

main()
```

Following this example, you can write the functions `diamonds` and `octagons` to draw four of each of those shapes, so our `main` becomes this:

```
import turtle

...

def main():
    turtle.setup()
    turtle.color("blue")
    triangles()
    turtle.color("red")
    diamonds()
    turtle.color("green")
    octagons()
    turtle.done()

main()
```

Remember that thing called abstraction? As an extra challenge, think about how to generalize the process of drawing multiple copies of a shape, each of a different size. Generalize the `triangles`, `diamonds` and `octagons` functions by replacing them with one that looks likes this, where sides is the number of sides and number is how many copies of the shape you want:

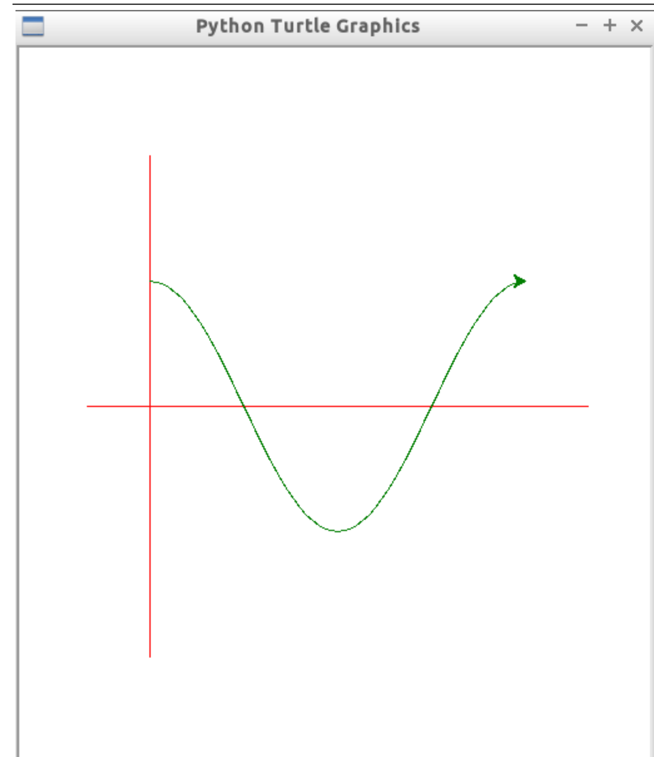
```
def multiples(sides, number):
```

Now we're pretty close to what the program is supposed to draw, all that's left is to move the turtle *with the pen up* before we draw each set of shapes. The turtle starts at position (0, 0) after `setup`, so we start by moving it a decent amount to the left before we draw the triangles:

```
import turtle

...

def main():
    turtle.setup()
    turtle.up()
    turtle.backward(150)
    turtle.down()
    turtle.color("blue")
    triangles()
    turtle.color("red")
    diamonds()
    turtle.color("green")
    octagons()
    turtle.done()

main()
```

Before calling `diamonds` we move it to the right (forward) by the same amount, and then again to the right by the same amount before calling `octagons`. Done!

Figure 2 Output of the `cosine.py` program.



2 Plotting Cosines [10 points]

The second program you will write plots one period of the cosine function from 0 to 2π. Please call your program `cosine.py` and nothing else! Figure 2 shows what the output of your program will look like.

You already know how to switch colors from the first program you wrote, so drawing the axes in red and the cosine curve in green should not be a problem. Here is some basic code to get you started (with the important functions missing of course):

```
import turtle

...

def main():
    turtle.setup()

    turtle.color("red")
```

```
    axes()

    turtle.color("green")
    plot()

    turtle.done()

main()
```

Instead of using `forward` and `left` to draw a certain line, it is more convenient for this program to be able to draw a line from the **current position** of the turtle to an **arbitrary** position on the screen. Luckily the `turtle` module has a `goto` function that does exactly that: If we are currently at position (-10, 10) and call `turtle.goto(100, 100)`, the turtle will draw a line (provided the pen is down!) from (-10, 10) to (100, 100)!

Let's attack the axes we need to draw first. Both axes should go from -200 to 200 in the respective coordinate, and they should cross at (-150, 0). Instead of writing the code for this twice, let's assume that there is a function `line(x1, y1, x2, y2)` that gets the x and y coordinates of **two** points to draw a line between (from the first point to the second point). If we have such a function, we can write the `axes` function as follows:

```
import turtle

...

def axes():
    line(-200, 0, 200, 0)
    line(-150, -200, -150, 200)

def main():
    turtle.setup()

    turtle.color("blue")
    axes()

    turtle.color("red")
    plot()

    turtle.done()

main()
```

You'll have to write the `line` function in terms of the `up`, `down`, and `goto` functions of the `turtle` module to finish drawing the axes.

Drawing the cosine wave itself requires a `for` loop to create the successive x values that we want to calculate the cosine for. You can access the cosine function using `turtle.cos(x)` where x is measured in **radians**, not degrees. If you look at Figure 2, we are obviously plotting the cosine function between 0 and 2π; however, in terms of screen coordinates, 0 is actually at -150 on our x-axis. Also, the result of `turtle.cos(x)` is always between -1 and 1 but the extrema in terms of screen coordinates are -100 and 100. So you will have to **shift** and **scale** the arguments to the cosine function (as well as the result you get back from it) suitably to make the plot have the right dimensions on the screen.

Set up the `for` loop inside your `plot` function so that the x coordinate takes the values from -150 to 150 in increments of 5. (Review the 3 argument version of the range function in order to do this.) So the first value will be -150, the next will be -145, the next will be -140, and so on, up to 150 at the other end. Inside the `for` loop, calculate the actual x value for the cosine function from these values; you'll have to divide and multiply with the correct factors to make this happen. When you get back the result, multiply it by the correct factor to make the extrema of the cosine -100 and 100. How do you achieve the plot itself? Just use the `goto` function in the right way!

3 Plotting Parametrics [10 points]

The third program you will write plots a *parametric curve*, a notion we'll explain briefly below. Please call your program `parametric.py` and nothing else! Figure 3 shows what the output of your program will look like. Note that the y axis crosses the x axis in the center this time, at position (0,0) in the graphics window.

When we talk about "plotting a function" we usually think of something like $y = \sin x$ where we determine the value for y from x and then plot the coordinates (x, y). A *parametric* curve of the kind shown in Figure 3 works a little differently: there is some parameter t and both the x **and** the y coordinate we plot depend on t. For example, here is the parametric curve you are supposed to plot:

$$x = \sin 2t$$

$$y = \cos 5t$$

The values for t range from 0 to 2π. Since we cannot write a `for` loop using floating point numbers, you'll have to write the range

Figure 3 Output of the `parametric.py` program.

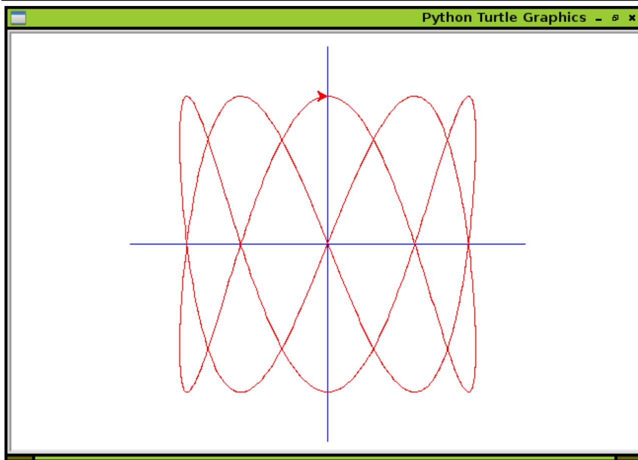
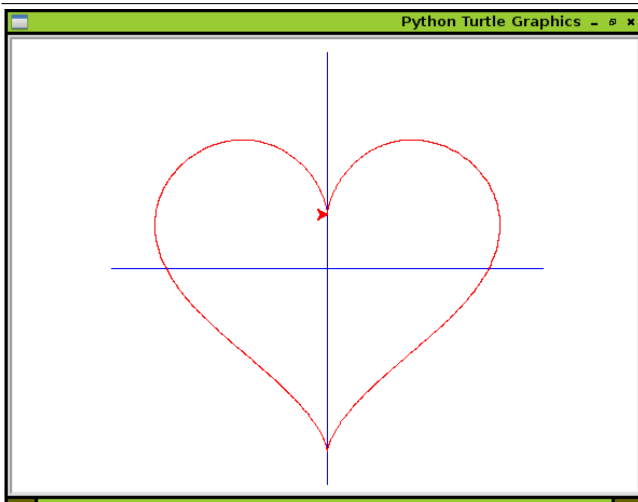


Figure 4 The parametric heart curve.



using integers. Use this one: `range(0, int(2*turtle.pi*1000), 10)` So instead of going from 0 to 2π we go from 0 to 2000π in terms of integer values; to get the correct floating point value, you'll have to divide t by 1000 before you use it in the parametric equations. Also, you need to scale the resulting x and y values by 150 to get the plot from Figure 3. Enjoy!

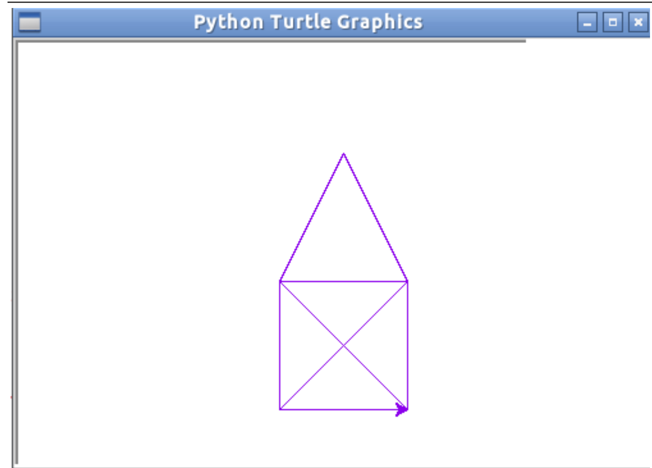
Bonus Curve

Parametric curves are fun because they can produce **lots** of interesting shapes. The parametric curve from above could, for example, be used in a game to move alien space ships across the screen in an interesting pattern. Figure 4 shows a parametric curve that could be useful for a love letter instead. The equation for the "heart curve" is as follows:

$$x = 16\sin^3 t$$

$$y = 13\cos t - 5\cos 2t - 2\cos 3t - \cos 4t$$

Figure 5 The house drawing challenge.



Once your program works for the basic parametric curve, it will be able to plot all of them. So feel free to play around and try to find a curve you like!

Final Challenge

As a final challenge, see if you can figure out how to draw the "house" in Figure 5 on a piece of paper without lifting the pen or retracing any lines. Once you have that down, write a python program to do it with the turtle. Use the online Python docs to look up some additional functions and features you can use with turtle graphics (`http://docs.python.org/2/library/turtle.html`). In particular, slow the turtle down so that you can really watch it draw the figure.