Eric Roberts
CSCI 121

Handout #5
September 5, 2018

Assignment #2—Simple Python Programs

Due: Wednesday, September 12

Your job in this assignment is to write programs to solve each of these problems. For the graphics problems, you need to download **`pgl.py`** from the class website.

Problem 1 (Chapter 2, exercise 5, page 73)

The ***digital root*** of a nonnegative integer *n* is defined as the result of summing the digits repeatedly until only a single digit remains. For example, the digital root of 1729 can be calculated using the following steps:

Step 1: 1 + 7 + 2 + 9 → 19
Step 2: 1 + 9 → 10
Step 3: 1 + 0 → 1

Because the total at the end of step 3 is the single digit 1, that value is the digital root. Write a Python function **`digital_root(n)`** that returns the digital root of **`n`**.

Problem 2 (Chapter 2, exercise 8, page 74)

Write a Python program that reads in integers up to a blank line and then prints both the largest and second-largest values in the user's input, as follows:

FindSecondLargest

```
This program finds the two largest integers.
Enter a blank line to stop.
 ? 223
 ? 251
 ? 317
 ? 636
 ? 766
 ? 607
 ? 607
 ?
The largest value is 766.
The second-largest value is 636.
```

The values in this sample run are the number of pages in the British hardcover editions of J. K. Rowling's *Harry Potter* series. The output tells us that the longest book is the *Harry Potter and the Order of the Phoenix* at 766 pages and the second-longest book is *Harry Potter and the Goblet of Fire* at 636 pages.

Problem 3

As far as I know, the only computer scientist ever to win a Pulitzer Prize is Douglas Hofstadter, whose marvelous book *Gödel, Escher, Bach* explores computer science in the style of Lewis Carroll. Hofstadter's book contains many interesting puzzles, many of which can be expressed in the form of computer programs. In Chapter XII, Hofstadter mentions a wonderful problem that is well within the scope of the control statements from Chapter 2. The problem can be expressed as follows:

> Pick some positive integer and call it *n*.
> If *n* is even, divide it by two.
> If *n* is odd, multiply it by three and add one.
> Continue this process until *n* is equal to one.

On page 401 of the Vintage edition, Hofstadter illustrates this process with the following example, starting with the number 15:

15	is odd, so I make 3*n*+1:	46
46	is even, so I take half:	23
23	is odd, so I make 3*n*+1:	70
70	is even, so I take half:	35
35	is odd, so I make 3*n*+1:	106
106	is even, so I take half:	53
53	is odd, so I make 3*n*+1:	160
160	is even, so I take half:	80
80	is even, so I take half:	40
40	is even, so I take half:	20
20	is even, so I take half:	10
10	is even, so I take half:	5
5	is odd, so I make 3*n*+1:	16
16	is even, so I take half:	8
8	is even, so I take half:	4
4	is even, so I take half:	2
2	is even, so I take half:	1

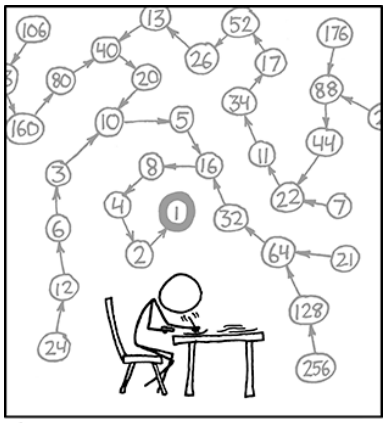
As you can see from this example, the numbers go up and down, but eventually—at least for all numbers that have ever been tried—comes down to end in 1. In some respects, this process is reminiscent of the formation of hailstones, which get carried upward by the winds over and over again before they finally descend to the ground. Because of this analogy, this sequence of numbers is usually called the ***Hailstone sequence,*** although it goes by many other names as well.

Write a function **`hailstone`** that takes an integer and then uses **`print`** to display the Hailstone sequence for that number, just as in Hofstadter's book, followed by a line showing the number of steps taken to reach 1. For example, your program should be able to produce a sample run that looks like this:

IDLE

```
>>> from hailstone import hailstone
>>> hailstone(17)
17 is odd, so I make 3n+1: 52
52 is even, so I take half: 26
26 is even, so I take half: 13
13 is odd, so I make 3n+1: 40
40 is even, so I take half: 20
20 is even, so I take half: 10
10 is even, so I take half: 5
5 is odd, so I make 3n+1: 16
16 is even, so I take half: 8
8 is even, so I take half: 4
4 is even, so I take half: 2
2 is even, so I take half: 1
The process took 12 steps to reach 1.
>>>
```

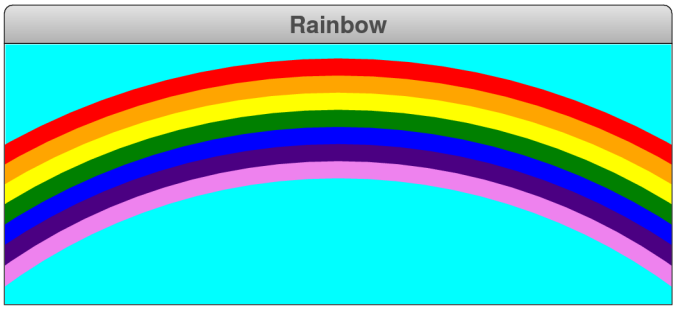
The fascinating thing about this problem is that no one has yet been able to prove that it always stops. The number of steps in the process can certainly get very large. How many steps, for example, does your program take when *n* is 27? The conjecture that this process always terminates is called the ***Collatz conjecture,*** and appears in the following *XKCD* cartoon by Randall Munroe:



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Problem 4 (Chapter 3, exercise 4, page 108)

Use the `GObject` hierarchy to draw a rainbow that looks something like this:



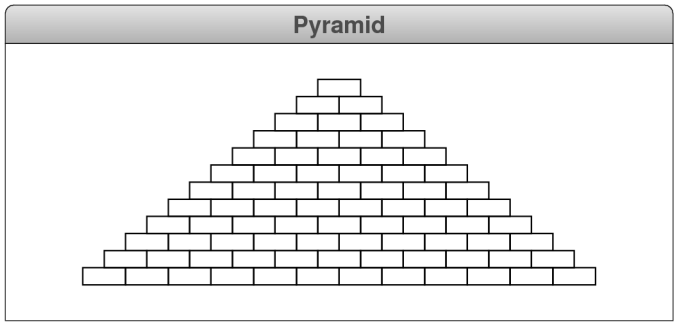
Starting at the top, the seven bands in the rainbow are red, orange, yellow, green, blue, indigo, and violet, respectively; cyan makes a lovely color for the sky. Remember that this chapter defines only the `GRect`, `GOval`, and `GLine` classes and does not include a graphical object that represents an arc. It will help to think outside the box, in a more literal sense than usual.

Rather than specify the exact dimensions of each circle (and there are indeed circles here), play around with their sizes and positioning until you get something that matches your aesthetic sensibilities. The only things we'll be concerned about are:

- The top of the arc should not be off the screen.
- Each of the arcs in the rainbow should get clipped along the sides of the window, and not along the bottom.

Problem 5 (Chapter 3, exercise 6, page 109)

Write a program that displays a pyramid on the graphics window. The pyramid consists of bricks arranged in horizontal rows, arranged so that the number of bricks in each row decreases by one as you move upward, as shown in the following sample run:



The pyramid should be centered in the window both horizontally and vertically. Your program should also use the following constants to make the program easier to change:

`BRICK_WIDTH`	The width of each brick
`BRICK_HEIGHT`	The height of each brick
`BRICKS_IN_BASE`	The number of bricks in the base