CSci 1133
Lab Exercise 4

Repetition

In this lab, you will continue to develop your understanding and use of conditional repetition in solving increasingly complex computational problems.

Note that from this point forward, lab exercises will (in general) have 4 categories of problems: *Warm-up*, *Stretch*, *Workout* and *Challenge*. The first three must be completed during the lab session. The "*Challenge*" problem(s) are optional for those of you who would like to further develop your skills or just can't get enough!

As you complete each of the following programs, you each need to commit your source code to your student GIT-hub repository. If you are working with one or more fellow students, make sure that each one of you commits the source code to your own repositories (respectively).

Warm-up

1). **Multiplication, Part I**

Write a pure function: `mul(a,b)` that will compute and return the product of two integers *a* and *b* using only a `while` loop and the addition operator (+). Your function should use the fewest possible number of iterations. (HINT: use the smaller of *a* and *b* to control the number of loop iterations)

2). **Multiplication, Part II**

The "Ancient Egyptian" multiplication algorithm is a more efficient way of performing multiplication by accumulating a sum. You can find a description of the algorithm here:

http://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication

The algorithm works as follows: given two integer values *a* and *b*, repeatedly multiply *a* by 2 and divide *b* by 2 until *b* is 0 (using *truncated* division!). At each iteration, if *b* is odd, add the current value of *a* to the accumulated sum. If it is even, do nothing. For example, multiplying 22 * 17:

a	*b*	*product*	
22	17	22	(*b* is *odd*, so *a* is added to the product)
44	8	22	(*b* is *even*, so nothing is added to the product)
88	4	22	
176	2	22	
352	1	374	

Write a pure function: `emul(a,b)` that will compute and return the product of two integers using the Ancient Egyptian algorithm as described. Assume both *a* and *b* are non-negative and make sure that your function runs in the fewest possible number of iterations based on the values of *a* and *b*.

(HINT: include a "print" statement for initial debugging that will show the values of a, b and the accumulating product as the loop progresses. Using print violates the idea of a *pure* function, but it can be removed after you verify that everything is working properly)

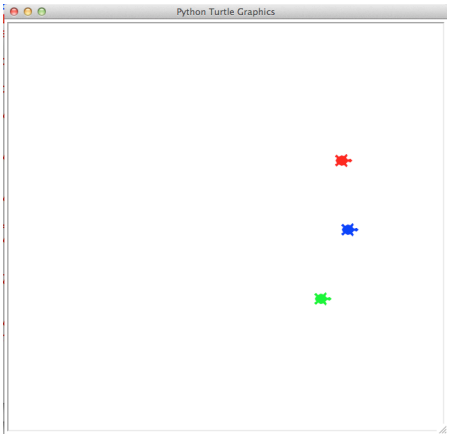
Stretch

1). **Complete Egyptian Multiplication**

Modify the `emul` function from the previous problem to handle any combination of positive or negative values for *a* and *b*. (HINT: perform the multiplication as if both multiplicands were positive, then assign the sign of the result depending on the original values)

2). **Off to the Races!**

Let's simulate a turtle race using turtle graphics. Write a well-structured Python program that uses a while loop to create a "race" between three individual turtles by moving each one forward using a random integer step from [1,15] across the window. First one to the edge wins! Here's an example of what the display should look like:



You will need to do a number of things, including:

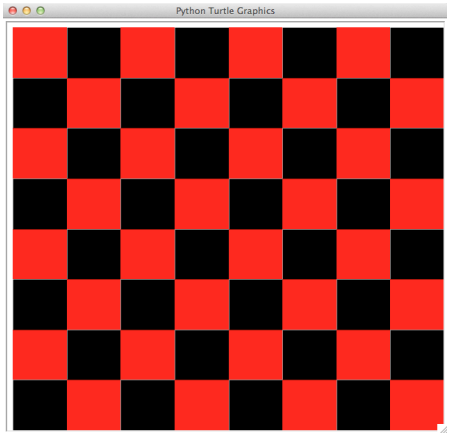
- Instantiate 3 separate turtle objects using the Turtle constructor (`turtle.Turtle()`)
- Position each turtle at the "starting line" (far left edge of window)
- Change each turtle shape to `'turtle'` (this *is* a turtle race after all!) and use a different color for each one.
- Continue the race until one of the turtles wins (reaches the far edge of the window)

HINT: You will find this easier if you first write a general function to move any turtle a random number of steps forward. This function requires that you supply the turtle object as an argument (can you explain why?). Do not use global variables!

Workout

1). **Checker Board**

Using turtle graphics, write a will structured Python program that will fill the turtle window with a red and black "checker board":



For this problem, you are required to use nested `while` loops to draw each row and column. You will find this much easier to accomplish by first constructing a general non-pure function (procedure) that draws a filled square given a color and x,y position. Call this function from your inner loop.

You will also find it helpful to first determine the window height and width at the start of your procedure and set the world coordinates based on the minimum dimension.

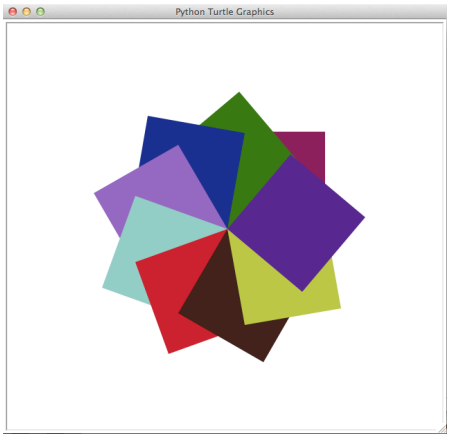
There are several things you can do to speed up the drawing:

- set the turtle speed to zero
- hide the turtle
- draw with the pen up (you don't need the outline, just the filled squares)

Challenge

1). **Spirograph**

Spirograph is a fun geometric drawing toy that uses the principle of repeatedly drawing regular figures inside of a circle to construct interesting patterns. Although the figures that the toy produces are mathematically complex, we can create a similarly fun game using turtle graphics by replicating regular polygons around a center point while choosing random colors. Here's an example created by rotating a square 9 times:



Write a well-structured Python program that will produce our simplified "spirograph" images. Your program should include a function: `spiro(sides, numberPerRev, radius)` that takes three arguments: the number of sides in the polygon, the number of polygons to uniformly draw in through one revolution, and the outside radius of each polygon in the figure. Note that the length of each side of a regular polygon is determined by its outside radius:

$$side = radius * 2 * \sin(180 / n)$$

where n is the number of sides in the polygon and the angle is in degrees.

To obtain a "random" color, do the following:

```
r = random.random()*.8
g = random.random()*.8
b = random.random()*.8
turtle.fillcolor(r,g,b)
```