CSci 1133
Lab Exercise 3

Program Control

"Flow" refers to the temporal nature of processes. A dynamic process evolves in time... it "flows" from one state to the next. *Sequential flow* in programming languages describes the automatic nature of execution: one statement at a time. The actions or calculations of one statement are completed prior to any subsequent statements.

Useful computations require the *alteration* of the sequential flow of operations depending on some *condition*. The two principal methods provided by imperative languages for controlling the flow of program execution are *conditional selection* ("if-else" decisions) and *conditional repetition* ("looping", "iteration"). *Conditional selection* in Python is accomplished using the `if`, `else` and `elif` statements, whereas *conditional repetition* employs a syntactically similar construct: the `while` statement.

In this lab exercise, we will further develop our understanding of *program control* mechanisms.

Primer on Pseudo-random Numbers

We often need to write programs that simulate *uncertain* outcomes such as the roll of dice, a hand of playing cards, samples drawn from some probabilistic distribution and so forth. Python provides a collection of interesting and useful functions that support the generation of samples from a *pseudo*-random sequence. You can think of *pseudo*-random numbers as the values that might come up if you rolled a 6-sided die over and over… The resulting sequence would be *random*… unpredictable. In fact, this is what a truly random variable represents (in the probabilistic sense). Computers, being entirely deterministic automatons, are incapable of generating *truly* random values. In fact, philosophers debate whether *truly* random events are even possible, but we'll save that for another day. First, let's see what functions are provided in the `random` module:

```
>>> import random <ENTER>
>>> help(random) <ENTER>
```

Yikes! There are a lot of interesting functions that all relate to the process of generating samples from a pseudo-random variable. To obtain a pseudo-random (seemingly-random) value you simply call the `random()` function:

```
>>> random.random() <ENTER>
```

Try it. You should get a floating-point number between 0 and 1. Now, try it again:

```
>>> random.random() <ENTER>
```

You get another value that is different than the last one. In fact, it is extremely difficult to predict the next number without "inside" knowledge of the function. For our purposes, we can consider each one a *random* value between 0 and 1, unique with respect to the values that preceded it. In other words, each time the `random()` function is called a different (unique) value will be returned.

Often, our programs will require pseudo-random *integers* in some range. For example, if we are asked to simulate the roll of a six-sided die, we would need uniformly generated pseudo random integers in the closed interval [1,6]. Python provides a useful function in the random module to accomplish this task.

```
>>> random.randint(1,6) <ENTER>
```

The `randint(a,b)` function will return a uniformly distributed pseudo-random *integer* from the closed interval [*a* , *b*].

Look at the function documentation for the `random()` function:

```
>>> help(random.random)  <ENTER>
```

As mentioned above, the `random()` function returns a floating-point value in the interval [0,1). If we need values from a wider range, we have two choices. We can *scale* the value by multiplying by some constant *K*, producing values in the range [0, *K*), or we can use the `uniform()` function in the random module. Look at the function documentation:

```
>>> help(random.uniform)  <ENTER>
```

The `uniform()` function allows us to specify the start and end values for a range of values that will be *uniformly* distributed. Try it a few times to obtain pseudo-random values in the range [-1, 1]:

```
>>> random.uniform(-1,1)  <ENTER>
>>> random.uniform(-1,1)  <ENTER>
>>> random.uniform(-1,1)  <ENTER>
```

1). **Rounding floating-point values**

In this weeks' readings, you learned about *explicit* type conversion in Python using the functions: `str()`, `float()` and `int()`. Note that if you convert a floating-point value to an integer using the `int()` function, the result will be *truncated*, not rounded as one might expect. e.g.,

```
>>> x = 4.67
>>> xi = int(x)
>>> print(xi)
 4
```

One way to round a floating-point number to the *nearest* integer (data type `int`) is to first add 0.5 if the number is positive or subtract 0.5 if the number is negative, then truncate (discard the fractional part of) the result.

a). Using conditional selection (if-else), write a pure function named `roundit` that will take a floating-point value as its only argument and return an *integer* rounded to the nearest integer value using the method described above.

b). Note that as your programs increase in complexity, it is essential to verify they work as intended. You need to carefully consider what set of input values should be tried in order to convince yourself that your program produces the correct answer in *all* cases. One important consideration is *path coverage*. A single branch *path* consists of the instructions that are executed when one "path" of an if-else statement is taken. This is dependent on the value of the conditional test. Path coverage involves providing inputs that ensure sure *every* branch path has been executed.

How many runs (test cases) should you make, at a minimum, to verify that this function is running correctly? What data should you input for each of the test cases and what is the expected result? **Discuss your answer with one of your TAs and demonstrate the result of your function for each test case.**

Procedural Decomposition and Well-structured Programs

Procedural decomposition is simply a fancy way of expressing the idea that large, complex problems are best solved by breaking them down into collections of simpler problems that are combined in some procedural way to solve the original (complex) problem. In computational problem solving, procedural decomposition results in the creation of a number of function definitions that can be *independently* designed, implemented and tested. It should be self-evident that simple functions are easier to design, write, debug and verify than complex programs, therefore procedural decomposition is a powerful and effective way to approach programming tasks in general.

The *structure* of a program refers to the way in which the various components (function definitions, variable and constant declarations, etc.) of a working program are organized. Although the structure of a program does not necessarily affect its execution, following a set of accepted conventions accrues a number of benefits:

- Programs that are easier for humans to comprehend (including the author!)
- Organized, "logical" approach to functionality
- Programs that are easier to describe and debug
- "Top-down" development approach that efficiently supports step-wise refinement

We will develop a better understanding of what a "well-structured" program looks like, but for now we will have you follow a few simple requirements for all the programs you write:

1. All variable names should help to "self-document" the program. In other words, variable names should be *descriptive* of what they are being used for! (e.g., account_balance, total, xLocation, etc.)
2. *All* of the executable lines of code in a program must appear within functions (i.e. there should not be loose bits of code floating around your program)
3. The *main* program code (the *starting* point of the program) must be included in a specific non-pure function named `main()` that does not return any value.
4. The only executable Python code allowed in the *global* name space (i.e., the area "outside" of a function definition) are constant declarations (e.g., PI = 3.114159, FEETPERMILE = 5280). Note that constants, by convention, use all capital letters.

There is a single exception to rule (2). Note that once you've placed the main part of your program in a `main()` function definition, you must somehow invoke the `main()` function in order for your program to run. This is easy in command-line mode; you simply type :

```
>>> main()<enter>
```

In *script* mode, however, you must add a single statement to the global namespace that will automatically cause the `main()` function to be called when the module is loaded and executed:

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

We will discover what this arcane bit of code does at a later time. For now, every program you write should contain a `main()` function definition with this statement appearing at the end of your module.

OK, let's get started. As you complete each of the following programs, make sure you 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). It is important for everyone to become familiar with the process of saving your code to GIT-hub!

2) **More Fun with Squares**

Using the `Drawsquare` function you created in Lab2, write a program that draws a series of identical squares in different orientations around a single point. Your program should use a `while` loop to draw 10 squares with sides of length 100, rotating the turtle by 36 degrees following each iteration.

Now modify your program to solicit the number of squares to draw from the user. You should put your main program code in a non-pure function named `main()` and invoke it using the

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

mechanism described earlier. Use the `turtle.numinput` function and determine the rotation angle (based on the user input) to rotate completely through 360 degrees.

3). **The Drunkards Walk**

A mathematical process known as a *random walk* is used to describe many physical and biological phenomena such as diffusion, Brownian motion, animal foraging, etc. A *random walk* is a formalization of a discrete process consisting of a series of unpredictable (random) steps.

A classic example of this process is known as the "Drunkards Walk": a drunken student leaves a pub in Dinkytown and attempts to walk back home. The individual is so inebriated however, that when he/she arrives at each street intersection a random direction is chosen (including a u-turn!). The question arises: "what is the probability that this poor student will ever arrive home before morning?"

Write a Python program using Turtle graphics that will visualize the Drunkard's Walk. Your program needs to do the following:

- Starting in the middle of the turtle window, obtain a random integer in the range [1,4] and change the turtle heading to 0, 90, 180 or 270 based on the value of the integer: 1, 2, 3 or 4 respectively.
- Move the turtle 20 steps in the new heading (with the pen down!)
- Using a while loop, repeat the process of randomly changing directions and moving until the turtle reaches the edge of the window (HINT: use the `xcor` and `ycor` turtle functions)
- Count the number of "steps" and, when the turtle reaches the window edge, terminate the while loop and display the total number of steps taken at the middle of the window. Use a font size large enough to read.

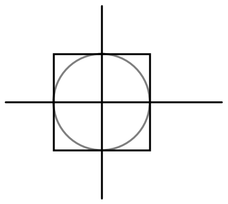
4). **A Slice of Pi**

Monte Carlo simulation methods employ large numbers of randomly generated sample values to determine solutions to complex numerical problems. This straightforward approach often succeeds when other methods prove difficult or intractable. In this exercise, we'll use a simple application of the *Monte Carlo* method to determine the value of *Pi*.

Consider a unit circle (a circle with radius = 1) centered at the origin of a Cartesian plane. The area of the circle is equal to the value *Pi*:

$$area = Pi * 1^2$$
$$area = Pi$$

If we construct a square such that the unit circle is exactly inscribed within its boundaries, each side of the square will have length 2 and range from [-1, 1] in both the *x* and *y* axes:



Note that the area of the bounding square is simply $2^2 = 4$.

Now, if we imagine "randomly" throwing a large number of darts at the square, a certain fraction of them will land *within* the circle. Assuming the locations of the randomly thrown darts are uniformly distributed, the ratio of those that land within the circle to the total number of darts thrown will be approximately equal to the ratio of the *area* of the circle to the square:

$$\frac{\text{darts-in-circle}}{\text{total-thrown}} = \frac{\pi}{4}$$

and the approximate value of Pi is simply obtained by multiplying the fraction of darts in the circle by 4. The accuracy of the approximation will improve as the number of darts increases.

The *Monte Carlo* method for this problem involves generating a large number of *x*, *y* coordinate points within the square [-1 <= *x* <= 1] , [-1 <= *y* <= 1] and counting those that fall within the circle. The ratio of those that fall within the circle to the total yields the probability we need.

Construct a Python simulation of this method using Turtle graphics. Write a program that will do the following:

- Use a while loop to simulate randomly "throwing" darts at a square. This is accomplished by doing the following in the body of the loop:
  - Obtain random values for both the x and y coordinates within the boundary of the square (i.e., [-1,1])
  - In order to visualize the process, you need to *scale* the x,y coordinates to span a larger region. Multiplying the x,y values by 150 will produce a plot region 300 pixels on a side.
  - Move the turtle to the scaled x, y coordinates and plot a single point using the `turtle.dot()` function (try a dot size of 5). The color of the dot will depend on whether the pseudo-random x,y coordinates are within the circle or not: use a green dot for points

within the unit circle and red for those that fall outside (HINT: recall our friend Pythagoras and his lovely theorem!)

- Try 1000 iterations and then display the estimated value of *Pi* using the `turtle.write()` function
- Here are a few helpful turtle functions that you might find useful (check the documentation):

`hideturtle()`	removes the turtle symbol from the display (not needed)
`speed()`	increase the speed of turtle movement
`delay()`	decrease the delay between writing operations
`penup()`	don't need to draw as you move!

If your simulation is sufficiently fast, you should try a larger number of iterations and see if the value of Pi gets closer to the value we expect.