CSci 1133, Spring 2015
Lab Exercise 4

Iterative Control

Last week we explored the use of Boolean expressions that conditionally select alternate program execution paths. This week, we continue the exploration of Boolean expressions and their use in dynamic facilities that support *conditional iteration*. *Conditional iteration* refers to the ability to repeatedly execute statements or groups of statements until some dynamically tested criteria is satisfied.

Python Modules

There is an abundance of functionality available in Python that has been developed and provided by other people. These useful *procedural abstractions* have been written in Python and supplied in the form of functions that we can simply use without needing to know the details of how they work. These functions are gathered into "modules" which we import in the same way as the `turtle` module explored in Lab1.

The `math` module is a particularly useful set of abstractions such as square root, sine, cosine, and other common mathematical functions. The `math` module also includes useful constant definitions such as pi. In order to use these functions and definitions, you must first *import* the `math` module into your program.

Start IDLE and type the following from the command-line shell:

```
>>> import math <ENTER>
```

If you ever forget which functions are included in a module, or are just curious what might be available, you can use the built-in Python function `help`:

```
>>> help("math") <ENTER>
```

The `help` function takes a string argument and provides a complete listing of the contents of the specified module. Try it in interactive mode. You can scroll up/down using the terminal controls.

Now let's try to compute the square root of a number using the `sqrt()` function from the math module:

```
>>> sqrt(144) <ENTER>
```

Oops! What happened? Importing a module results in the creation of a "module" object. Each function in the module is considered a *method* of the module object. We'll learn more about object methods as we proceed, but for now you can consider them to be functions that are closely associated with some object. In order to invoke the object's method, you must first dereference it from the module name using the *dereferencing operator* `'.'` like this:

```
>>> math.sqrt(144) <ENTER>
```

Go ahead and try it.

That's more like it! You always need to provide the object name followed by the period before the method name.

There is another way to import the functions in a module without creating a module object:

```
>>> from math import * <ENTER>
```

This will import *all* the function definitions from the module without creating a module object (the * symbol means 'all' in this context). If we do this, we can simply invoke the square root using the function name in the way we tried earlier:

```
>>> sqrt(144) <ENTER>
```

But be careful! If you inadvertently import a function with the same name as a function you've already defined yourself, *your* function definition will be overwritten by the *module* function definition. For this reason it is generally safer (and better) to import module functions as methods to avoid naming conflicts.

Pseudo-random Numbers

We will need to write programs that simulate *uncertain* outcomes such as the roll of dice, the dealing of playing cards, samples drawn from some 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:

```
>>> 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. Next, go ahead and import the `random` module:

```
>>> import random <ENTER>
```

To obtain a pseudo-random (seemingly-random) value you simply call the `random()` method of the `random` module:

```
>>> 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 simulating the roll of a six-sided die, we would need uniformly generated (pseudo) random integers in the range [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*].

Warm-up

1). **Celsius/Fahrenheit Crossover**

Write a Python program that finds the temperature, as an *integer*, that is the same in both Celsius and Fahrenheit. The formula to convert from Celsius to Fahrenheit is:

$$Fahrenheit = \frac{9}{5}Celsius + 32$$

Your program should create two integer variables for the temperature in Celsius and Fahrenheit. Initialize the temperature to 100 degrees Celsius. In a loop, decrement the Celsius value and compute the corresponding temperature in Fahrenheit until the two values are the same. Hint: you will need to avoid floating-point division or use explicit type conversions!

Stretch

1). **Take-Away Game**

Take-Away is one variant of an ancient game called *Nim* that involves two players alternately removing 1, 2 or 3 objects (coins, chips, matchsticks, etc.) from a pile of 21 objects. The player who plays last wins. The objective is to play in such a way that there are 1, 2 or 3 objects remaining when it is your turn so that you can immediately remove them all and win. In this version of the game, it turns out that the first player will always win if he/she plays according to a simple algorithm.

Note that the objective is to ultimately force your opponent to pick from a pile of 4 objects, which will immediately lead to a win (because the opponent must leave 1,2 or 3!). Therefore we want the pile to be 5, 6 or 7 objects when it is our turn. In order to have 5,6 or 7 on our turn, we want the pile to be 8 for the opponent's turn. Working "backwards" in this way, we can see that the simple algorithm is to always force the opponent to choose from a number of objects that is divisible by 4.

Write a Python program to play the Take-Away game. Your program should do the following:

1. Allow the opponent the choice of going first or second.
2. Check all input to ensure it is within bounds. If not, produce a suitable error message and continue to solicit input until a valid value is provided.
3. Implement the algorithm described above using a suitable loop.
4. Determine the "winner" and provide an appropriate output message.

Here's a sample of what your output "might" look like:

```
would you like to go first (y/n)? n
ok, I'll take 1
```

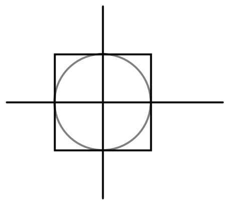
```
20 objects remain, choose 1,2 or 3: 2
18 objects remain, I'll take 2
16 objects remain, choose 1,2 or 3: 3
13 objects remain, I'll take 1
12 objects remain, choose 1,2 or 3: 3
9 objects remain, I'll take 1
8 objects remain, choose 1,2 or 3: 3
5 objects remain, I'll take 1
4 objects remain, choose 1,2 or 3: 3
1 objects remain, I'll take 1
I win.  Thanks for playing


would you like to go first (y/n)? y
21 objects remain, choose 1,2 or 3: 1
20 objects remain, I'll take 1
19 objects remain, choose 1,2 or 3: 3
16 objects remain, I'll take 1
15 objects remain, choose 1,2 or 3: 3
12 objects remain, I'll take 1
11 objects remain, chose 1,2 or 3: 3
8 objects remain, I'll take 1
7 objects remain, choose 1,2 or 3: 3
4 objects remain, I'll take 1
You win!


would you like to go first (y/n)? n
ok, I'll take 1
20 objects remain, choose 1,2 or 3: 0
oops, you must choose 1,2 or 3...

20 objects remain, choose 1,2 or 3:
```

Workout

1). **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 square with sides of length 2*r* = 2. Now consider a circle, exactly inscribed within the bounding square. This circle will have radius equal to *r* = 1 and its *area* will be exactly the value Pi:



If we "randomly" throw a dart at the square, the *probability* that it will fall within the inscribed circle is equal to the ratio of the circle area to the area of the square:

$p($ *dart_in_circle* $) = \text{Pi} / 4$

If we throw a large number of random "darts" at the square, the proportion of those "in" the circle to those "outside" the circle will come very close to the probability: *p*(*dart_in_circle*). Multiplying this probability by 4 yields the value we are seeking:

$p($ *dart_in_circle* $) * 4 = \text{Pi}$

Using 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.

Write a program that will estimate the value of Pi using the *Monte Carlo* method described above. Recall the parametric form of a circle: $x^2+y^2 = r$ and determine those points that fall *within* this circle. Try various sample sizes, including 100, 1,000, 100,000 and 1,000,000 and observe how the sample size affects the accuracy of the result.

2). **Another Slice of π**

Another way to approximate of the value of Pi is provided by the series:

$$\pi = \frac{6}{\sqrt{3}}\left(1 - \frac{1}{3^1 \cdot 3} + \frac{1}{3^2 \cdot 5} - \frac{1}{3^3 \cdot 7} + \cdots\right)$$

Write a program to compute π that proceeds as follows: first the user should input a tolerance. Then the program should compute and display the approximation to π using just one term from the series, then using just two terms in the series, then just three terms, etc. It should do this until the absolute value of the difference between two successive approximations is less than the tolerance.

Output both the computed value of π and the number of terms that were required.

Hint: Note how the exponent and multiplier in each term's denominator changes. Also note that each term is either positive or negative. You can easily compute this by taking -1 to an increasing integral power which yields the series:

1, -1, 1, -1, 1, -1, 1, . . .

Challenge

Try these challenge problems if you have extra time or would like additional practice outside of lab.

1) **Roman Numerals**

The ancient Roman numbering system employed a sequence of letters to represent numeric values. Values were obtained by summing individual letter values in a left-to-right fashion. The value of individual Roman numerals is provided in the following list:

I	= 1
V	= 5
X	= 10
L	= 50
C	= 100
D	= 500
M	= 1000

In Roman numbers, larger numeral values generally appear before smaller values, and no single numeral may be repeated more than 3 times in a row. There are a few numbers that cannot be represented with these restrictions, so an additional rule was provided to deal with these cases: if a higher value numeral is preceded by a *single* smaller numeral, the smaller value is subtracted from the larger one. This rule is only applied to a small number of cases: IV (4), IX (9), XL (40), XC (90), CD (400) and CM (900). Using these simple rules, we can construct a method to convert an integer (decimal) value to a Roman number as follows:

Step 1:

If the value of the number is is in [900,999] output "CM"
If the value is in [500,899] output "D"
If the value is in [400,499] output "CD"
If the value is in [100,399] output "C"
If the value is in [90,99] output "XC"
If the value is in [50,89] output "L"
If the value is in [40,49] output "XL"
If the value is in [10,39] output "X"
If the value is 9, output "IX"
If the value is in [5,8] output "V"
If the value is 4, output "IV"
If the value is in [1,3], output "I"

Step 2:

Subtract the value of the 1 or 2 letter sequence that was just output from the integer number value, producing a 'remainder'

Step 3:

Repeat the process until the remaining value is zero.

Write a program that will convert a positive integer between 1 and 999 to its Roman Numeral equivalent. Your program needs to do the following:

- Input an integer value between 1 and 999 and check to make sure the input value is within acceptable bounds. If not, output an error message and use an indefinite loop to continue re-soliciting input until a valid number is entered. After a maximum of 4 attempts, end the program if a valid number has not been entered.

- Convert the integer value to a Roman number to a string (Hint: use concatenation) of consecutive letters without any spaces (ie. XCIII)

Example:

```
Enter an integer value from 1 to 999: 0
Invalid input.
Enter an integer value from 1 to 999: 42
Roman numeral equivalent: XLII

Enter an integer value from 1 to 999: 3
Roman numeral equivalent: III

Enter an integer value from 1 to 999: 9
Roman numeral equivalent: IX

Enter an integer value from 1 to 999: 999
Roman numeral equivalent: CMXCIX
```

2). **Factors of an Integer**

Write a Python program that will read in any positive integer value and display it as a product of its *smallest* integer factors in *increasing* order.

Examples:

```
Input a positive integer: 8
Factors: 2*2*2

Input a positive integer: 25
Factors: 5*5

Input a positive integer: 80
Factors: 2*2*2*2*5

Input a positive integer: 17
Factors: 17
```