A Slight Detour: Random Walks

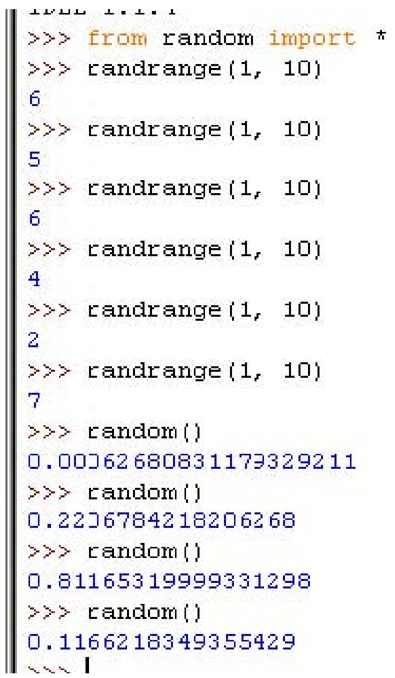
One way you can do interesting things with a program is to introduce some randomness into the mix. Python, and most programming languages, typically provide a library for generating random numbers. Generating random numbers is an interesting process in itself but we will save that discussion for a later time. Random numbers are very useful in all kinds of computer applications, especially games and in simulating real life phenomena. In order to access the random number generating functions in Python you have to import the ***random*** library:

```
from random import *
```

There are lots of features available in this library but we will restrict ourselves with just two functions for now: **`random`** and **`randrange`**. These are described below:

- **`random()`**: returns a random number between 0.0 and 1.0.
- **`randrange(A, B)`**: returns a random number in the range `[A…B-1]`.

Here is a sample IDLE window illustrating the use of these two functions:



```
>>> from random import *
>>> randrange(1, 10)
6
>>> randrange(1, 10)
5
>>> randrange(1, 10)
6
>>> randrange(1, 10)
4
>>> randrange(1, 10)
2
>>> randrange(1, 10)
7
>>> random()
0.0036268083117932911
>>> random()
0.2236784218206268
>>> random()
0.81165319999331298
>>> random()
0.11662183493555429
>>>
```

Do This: When we first learned about graphics and drawing we learned that several colors have been assigned names that can be used to select colors. You can also create colors of your own by specifying their ***red***, ***green***, and ***blue*** values. Each color is made up of 3 values, called **RGB** or red, green, and blue color values. Each of these values in the in the range 0..255 and is called a 24 bit color value (why?). With this scheme, the color with the values (255, 255, 255) (i.e., red = 255, green = 255, and blue = 255) is white; (255, 0, 0) is pure red, (0, 255, 0) is pure blue, (0, 0, 0) is black, (255, 175, 175) is a pink, etc. You can have as many as 256 x 256 x 256 colors (i.e., over 16 million colors!). Given specific RGB values, you may create any new color using the **graphics** library command **`color_rgb`**:

```
myColor = color_rgb(255, 175, 175)
```

We will be talking about colors a lot more over the next few weeks, but for now, just try it out (and the random number generator) by examining and implementing the program below:

```
# Program to draw a bunch of random colored circles

# Required Libraries
from random import randrange
from graphics import *

# creates a Circle centered at point (x, y) of radius
def makeCircle(x, y, r):
    return Circle(Point(x, y), r)

# creates a new color using random RGB values
def makeColor():
    red = randrange(0, 256)
    green = randrange(0, 256)
    blue = randrange(0, 256)
    return color_rgb(red, green, blue)

# Create and display a graphics
def main():
    width = 500
    height = 500

    myWindow = GraphWin('Circles', width, height)
    myWindow.setBackground('white')

    # draw a bunch of random circles with random colors.
    N = 500
    for i in range(N):
        # pick random center point and radius in the window
        x = randrange(0,width)
        y = randrange(0,height)
        r = randrange(5, 25)
        c = makeCircle(x, y, r)

    # select a random color
    c.setFill(makeColor())

    # draw the circle
    c.draw(myWindow)

  main()
```

BUILDING BIGGER PROGRAMS: SCOPE

Notice our use of functions to organize the circle program. From a design perspective, the two functions `makeCircle` and `makeColor` are written differently. This is just for illustration purposes. You could, for instance, define `makeCircle` just like `makeColor` *so it doesn't take any parameters* and generates the values of x, y, and radius as follows:

```
def makeCircle():
    # creates a Circle centered at point (x, y) of radius r
    x = randrange(0,width)
    y = randrange(0,height)
    r = randrange(5, 25)
    c = makeCircle(x, y, r)
   return Circle(Point(x, y), r)
```

Unfortunately, as simple as this change seems, the function is not going to work. In order to generate the values of x, and y it needs to know the width and height of the graphics window. But width and height are defined in the function main and are not available or accessible in the function above. This is an issue of ***scope*** of names in a Python program: what is the scope of accessibility of a name in a program?

Python defines the scope of a name in a program textually or lexically. That is, any name is visible in the text of the program/function *after* it has been defined. Note that the notion of *after* is a textual notion. Moreover, Python restricts the accessibility of a name to the text of the function in which it is defined. That is, the names width and height are defined inside the function main and hence they are not visible anywhere outside of main. Similarly, the variables `red`, `green`, and `blue` are considered local to the definition of `makeColor` and are not accessible outside of the function, `makeColor`. So how can `makeCircle`, if you decided it would generate the `x` and `y` values relative to the window size, get access to the width and height of the window? There are two solutions to this. First, you can pass them as parameters. In that case, the definition of `makeCircle` will be:

```
def makeCircle(w, h):
    # creates a Circle centered at point (x, y) of radius r
    # such that (x, y) lies within width, w and height, h
    x = randrange(0,w)
    y = randrange(0,h)
    r = randrange(5, 25)
    c = makeCircle(x, y, r)
    return Circle(Point(x, y), r)
```

Then the way you would use the above function in the main program would be using the command:

```
C = makeCircle(width, height)
```

That is, you pass the values of `width` and `height` to `makeCircle` as parameters. The other way to define `makeCircle` would be exactly as shown in the first instance:

```
def makeCircle():
    # creates a Circle centered at point (x, y) of radius r
    x = randrange(0,width)
    y = randrange(0,height)
    r = randrange(5, 25)
    c = makeCircle(x, y, r)
    return Circle(Point(x, y), r)
```

However, you would move the definitions of width and height outside and before the definitions of all the functions:

```
from graphics import *
from random import randrange
width = 500
height = 500
def makeCircle(x, y, r):
…
def makeColor():
…
def main():
…
```

Since the variables are defined outside of any function and before the definitions of the functions that use them, you can access their values. You may be wondering at this point, which version is better? Or even, why bother? The first version was just as good. The answer to these questions is similar in a way to a paragraph is an essay. You can write a paragraph in many ways. Some versions will be more preferable than others. In programming, the rule of thumb one uses when it comes to the scope of names is: ensure that only the parts of the program that are supposed to have access to a name are allowed access. This is similar to the reason you would not share your password with anyone, or your bank card code, etc. In our second solution, we made the names `width` and `height` ***globally*** visible to the entire program that follows. This implies that even `makeColor` can have access to them whether it makes it needs it or not.

You may want to argue at this point: what difference does it make if you make those variables visible to `makeColor` as long as you take care not to use them in that function? You are correct, it doesn't. But it puts an extra responsibility on your part to ensure that you will not do so. But what is the guarantee that someone who is modifying your program chooses to? We used the simple program here to illustrate simple yet potentially hazardous decisions that dot the landscape of programming like land mines.

Programs can crash if some of these names are *mishandled* in a program. Worse still, programs do not crash but lead to incorrect results. However, at the level of deciding which variables to make local and which ones to make global, the decisions are very simple and one just needs to exercise safe programming practices.

MAKING DECISIONS

There is a famous theorem in computer science that says that if a device is capable of being programmed to do *sequential execution*, *decision making*, and *repetition*, it is capable of expressing any computable algorithm. This is a very powerful idea that is encapsulated in terms of very simple ideas. In other words, the theorem argues that, if you can describe a solution to a problem in terms of a combination of those three executable models then you can get a computer to solve that problem! We have already looked at two of the three models (sequential execution and repetition). Now it is time to look at decision making!

Let's Talk About This, That, and The Other

The overall goal of decision making is to alter the flow of a program by conditionally controlling the execution of a set of commands based upon whether or not a condition is met. The simplest way to do this is through the use of an **`if`-statement**. For example, the following statement compares two numbers and prints the value of a if it is larger:

```
if (a > b):
   print a
```

In the above code block, we are also introducing another type of control flow statement, the **`if`**-statement. This statement enables simple decision making inside computer programs. The simplest form of the `if`-statement has the following structure:

```
if <CONDITION>:
   <do something>
   <do something>
   ...
```

That is, if the condition specified by `<CONDITION>` is `True,` then whatever is specified in the body of the `if`-statement is carried out. In case the `<condition>` is `False`, all the statements under the `if` command are skipped over.

The `if`-statement is a way of making simple decisions (also called ***one-way decisions***). That is, you can conditionally control the execution of a set of commands based on a single condition. The `if`-statement is Python is quite versatile and can be used to make two-way or even multi-way decisions. Here is how you would use it to choose among two sets of commands (i.e., create branch or fork in the program):

```
if <condition>:
   <this>
else:
    <that>
```

That is, if the `<condition>` is true it will do the commands specified in `<this>`. If, however, the `<condition>` is false, it will do `<that>`. For example, the following function compares two numbers and returns the larger of the two:

```
def findMax(a, b):
    if (a > b):
          return a
    else:
     return b
```

Similarly, you can extend the `if`-statement to help specify multiple options:

```
if <condition-1>:
    <this>
elif <condition-2>:
    <that>
elif <condition-3>:
    <something else>
...
...
else:
    <other>
```

Notice the use of the word **`elif`** (yes, it is spelled that way!) to designate ***'else if'***. Thus, depending upon whichever condition is true, the corresponding `<this>`, `<that>`, or `<something else>` will be carried out. If all else fails, the `<other>` will be carried out.

DECISION MAKING IN PRACTICE: ROCK, PAPER, SCISSORS!

Let's look at decision making in a bit more detail by exploring a classic game playing situation: Rock, Paper, Scissors!

In this game, two players play against each other. At the count of three, each player makes a hand gesture indicating that they have selected one of the three items: paper (the hand is held out flat), scissors (two fingers are extended to designate scissors, or rock (indicated by making a fist). The rules of deciding the winner are simple: if both players pick the same object it is a draw; otherwise, paper beats rock, scissors beat paper, and rock beats scissors. Let us write a computer program to pay this game against the computer. Here is an outline for one round of the game:

```
# Computer and player make a selection
# Decide outcome of selections (draw or who won)
# Inform player of decision
```

If we represent the three items in a list, we can have the computer pick one of them at random by using the random number generation facilities provided in Python. For example, if the items are represented as:

```
items = ['Rock', 'Paper', 'Scissors']
```

Then we can select any of the items above as the computer's choice using the following statement:

```
# Computer makes a selection
myChoice = items[<0 or 1 or 2 selected randomly>]
```

That is `items[0]` represents the choice `'Rock'`, `items[1]` represents `'Paper'`, and `items[2]` is `'Scissors'`.

As explained in the previous section, a random number in any range can be generated using the `randrange` function from the `random` library module in Python. Thus we can model the computer making a random selection using the following code block:

```
from random import randrange

# Computer makes a selection
myChoice = items[randrange(0, 3)]
```

Recall that `randrange(n, m)` returns a random numbers in the range `[n..m1]`. Thus, `randrange(0, 3)` will return either 0, 1, or 2. And the computer has successfully chosen whether it is going to play a Rock, Paper, or Scissors.

But Rock, Paper, Scissors is a two player game and so now, it is the user's turn. We can use the `input` command to ask the player to indicate his/her selection. :

```
# Player makes a selection
yourChoice = input('Please enter Rock, Paper, or Scissors: ')
```

Now that we know how to the computer and player make their selection, we need to think about deciding the outcome. Here is an outline:

```
if both picked the same item then it is a draw
if computer wins then inform the player
if player wins then inform the player
```

Rewriting the above algorithm in Python using `if`-statements we can come up with a pseudocode first draft:

```
if myChoice == yourChoice:
   print 'It is a draw.'
if <myChoice beats yourChoice>:
   print 'I win.'
```

```
else:
   print 'You win.'
```

Now all we need to do is figure out how to write the condition `<myChoice beats yourChoice>`.

The condition has to capture the rules of the game mentioned above. We can encode all the rules in a conditional expression as follows:

```
if (myChoice == 'Paper' and yourChoice == 'Rock')or
    (myChoice == 'Scissors' and yourChoice == 'Paper')or
    (myChoice == 'Rock' and yourChoice == 'Scissors'):
    print 'I win.'
  else:
     print 'You win.'
```

The conditional expression above captures all of the possibilities that should be examined in order to make the decision. But it is a little confusing. We could write this another way, that takes more lines of code but is a bit more readable:

```
if myChoice == 'Paper' and yourChoice == 'Rock':
  print 'I win.'
elif myChoice == 'Scissors' and yourChoice == 'Paper':
  print 'I win.'
elif myChoice == 'Rock' and yourChoice == 'Scissors':
  print 'I win.'
else:
  print 'You win.'
```

That is each condition is examined in turn until one is found that confirms that the computer wins. If none such condition is true, the `else`-part of the `if`-statement will be reached to indicate that the player won.

Another alternative to writing the decision above is to encode the decision in a function. Let us assume that there is a function `beats` that returns `True` or `False` depending on the choices. We could then rewrite the above as follows:

```
if myChoice == yourChoice:
   print 'It is a draw.'
if beats(myChoice, yourChoice):
   print 'I win.'
else:
   print 'You win.'
```

Let us take a closer look at how we could define the `beats` function. It needs to return `True` if `myChoice` beats `yourChoice`. So all we need to do is encode the rules of the game described above. Here is a draft of the function:

```
   # Does me beat you? If so, return True, False otherwise.
   def beats(me, you):
       if me == 'Paper' and you == 'Rock':
          # Paper beats rock
          return True

       elif me == 'Scissors' and you == 'Paper':
          # Scissors beat paper
          return True
       elif me == 'Rock' and you == 'Scissors':
          # Rock beats scissors
          return True

       else:
          return False
```

Once again, we have used the `if`-statements in Python to encode the rules of the game. Now that we have completely fleshed out all the critical parts of the program, we can put them all together as shown below:

```
# A program that plays the game of Rock, Paper, Scissors!

from random import randrange

# Does me beat you? If so, return True, False otherwise.
def beats(me, you):
   if me == 'Paper' and you == 'Rock':
      # Paper beats rock
      return True

   elif me == 'Scissors' and you == 'Paper':
      # Scissors beat paper
      return True

   elif me == 'Rock' and you == 'Scissors':
      # Rock beats scissors
      return True

   else:
      return False

def main():# Play a round of Rock, Paper, Scissors!
    print 'Let us play Rock, Paper, Scissors!'

    # define items
    items = ['Rock', 'Paper', 'Scissors']

    # Computer and Player make their selection...
    # Player makes a selection
    yourChoice = input('Please enter Rock, Paper, or Scissors: ')
    # Computer makes a selection
    myChoice = items[randrange(0, 3)]

    # inform Player of choices
    print 'I picked', myChoice
    print 'You picked', yourChoice

    # Decide if it is a draw or a win for someone
    if myChoice == yourChoice:
        print 'We both picked the same thing.'
        print 'It is a draw.'
    elif beats(myChoice, yourChoice):
        print 'Since', myChoice, 'beats', yourChoice, '...'
        print 'I win.'
    else:
        print 'Since', yourChoice, 'beats', myChoice, '...'
        print 'You win.'
    print 'Thank you for playing. Bye!'

main() # invoke the program
```

A few more print commands were added to make the interaction more natural.

EXERCISES

1. Implement and modify the Rock, Paper, Scissors program above and play it several times to make sure you understand it completely. Modify the program so that it plays several rounds and the user is able to choose when to stop playing. Implement a scoring system that keeps track of the number of times each player has won and the number of draws. When the user is done playing, report the score. 15 pts.

For the following questions: make all changes to the `histogram.py` file that is downloadable from the course website. Submit your final version (i.e., what you have after completing #5) as the answer to question 2 – 5.

Download the **`histogram.py`** file from the website. This file provides a template for the large-scale histogram program that we are be writing. Examine the file closely. Make sure you understand the **`main()`** program:

```
Line 1: def main():
Line 2:     # IMPORTANT VALUES
Line 3:     # data set
Line 4:     data = [<a really long list of values here...>]

Line 5:     # distance axes are offset from edge of window
Line 6:     axesOffset = 20

Line 7:     # prompt user for number of bins
Line 8:     numBins = input('Please enter number of bins: ')
Line 9:      binSize = calcBinSize(data, numBins) # calculate bin size

Line 10:     xLength = numBins * binSize # the length of the x-axis

Line 11:     # Create the Window

Line 12:     # Draw the Axes

Line 13:     # Calculate Frequency Data

Line 14:     # Print Frequency Data to Screen

Line 1: main() # invoke main
```

Please note the following: in the `histogram.py` file, Line 4 defines a long list of values, or the **`data`** with which we will be working. Line 9 calls the function that you wrote to calculate the size of a bin. You will cut and paste that code into the histogram file (see Exercise 2). Please modify you functions as necessary to match the naming conventions and order of parameters listed in `def`-statement for each in the `histogram.py` file. This will make it easier to assemble the whole program. Lines 11 -14 are place markers indicating modifications to the program you will make when completing Exercises 3 – 5. As you build your histogram program, keep in mind the issue of scope and how information is passed from one part of the program to another.

2. Copy and paste the functions we have already completed: finding bin sizes (**`calcBinSize`**), drawing a bar (**`drawBar`**), and calculating relative frequency data (**`calcRelFreq`**). If you were unable to solve these problems on your own, please feel free to use the solutions from the key.

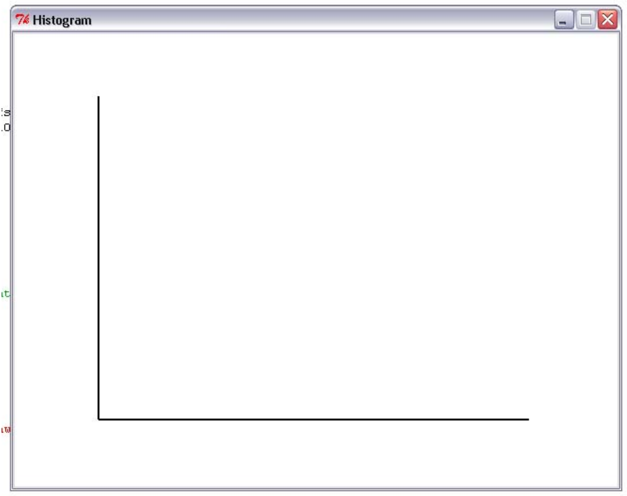
3. Fill in the **`createWindow`** function. (6 pts)

   The function should do the following:

   a. Create a graphics window that is 640 x 480 pixels. The title of the window should be 'My Histogram'.
   b. Make the background of the window white.
   c. Sets the coordinate system so that it is easy to draw (i.e., lower left is (0,0)). The upper right should have a height of the window is 100 + 2 * offset (why?) and a width of the xLength + 2 * offset.

4. Fill in the **`drawAxes`** function so that the x- and y-axes are drawn. Make sure to offset the axes from the edge of the window (see figure below). Make the line widths = 2. Update your **`main()`** program so that it executes the **`createWindow`** and **`drawAxes`** functions. (6 pts)

   When you run the program and enter any value for the number of bins, the program should produce the following graphic:



5. Fill in the **`calcFrequency`** function. This function should take the raw data and calculate and return frequency data. A variable, **`data`** has been defined in the **`main()`** program that you can use as an argument to the function.

   Below are points that we highlighted in our class discussion.

   HINT: The frequency of a bin = count of number of observations that fall into the bin.

   HINT: For the purpose of this exercise, a value is in a bin if the value >= binMin and value < binMax, where binMin and binMax are the lower and upper ranges of the bin, respectively.

   HINT: binMin of the first bin is the minimum value in the list.

   HINT: binMax = binMin + binSize

   HINT: The algorithm we came up with in class is listed below. Please note that that some of these tasks may involve multiple steps and that there may be steps you need to take that are not explicitly listed (e.g., setting initial values for your loops, updating values of binMin and binMax as you iterate through the bins).

```
1. Create an empty list of zeros to store frequencies calculated
   for each bin
2. For each bin do the following:
      a. For each value in the data list do the following:
           i.  check to see if the value is in the current bin
                 1. if it is, add one to the frequency count for
                    the current bin
3. Return the list of frequencies
```

Modify the **`main()`** program so that it calculates and prints out the frequency of the variable **`data`**. Run your program and test for 5, 10, and 20 bins. You should get the following:

```
5 Bins:[31, 19, 16, 22, 12]
10 Bins:[13, 18, 7, 12, 11, 5, 6, 16, 6, 6]
20 Bins:[7, 10, 8, 4, 3, 6, 6, 7, 4, 5, 0, 6, 0, 10, 6, 1, 5, 4, 2]
```

(25 pts)