Lab 1: Random Walk and *Spyder*

Related sections of the book

- Chapter 2; this chapter is available online at http://www-personal.umich.edu/~mejn/cp/chapters/programming.pdf
- There are some conventions we will use in our code that differ slightly from what is in the book.

Also remember that Google is your friend, and you can talk to other people in the class, some of whom have a lot of programming experience.

Objectives

By the end of this lab you should have

1. a working Python program that computes a "random walk" for a particle and plots its path on the screen.
2. Understand essential commands in Python.

Prelab Questions

Read the rest of this lab and only then answer the following questions:

1. Notice that in this program you will be importing a variety of commands. Examine what those commands do by using google or your editor's built-in help (e.g. - To see what empty does, look up 'numpy.empty', to see what plot does, lookup 'matplotlib.pyplot.plot'). Write down what these commands will do!
2. Why are you asked to define a variable **`n_timesteps`** with a constant value of 100 instead of just typing "100" when you need a value of 100 in the code?
3. Outline what the function **`coord(step_size)`** needs to do to return ±**`step_size`**. Describe it in English, not runnable code.

Program Description: Random Walk

We will define a random walk in the following way:

- A particle's position is given by two coordinates x and y.
- The particle starts at the origin, $x = 0,\ y = 0$.
- After an interval of time called a time step dt, the particle may jump by an amount $dx = \pm step\_size$ and $dy = \pm step\_size$ so that the new position is $x_{new} = x + dx$ and $y_{new} = y + dy$.
- **Note:** In the code you write today the variables **`x`** and **`y`** will each be an array containing all of the steps taken, not a single x-y position.

Part 1: Writing a Program to Perform a Random Walk

You first step is to write a program that will start a single particle at the origin and perform a random walk of 100 steps, printing out the final position. to accomplish this, do the following:

In your Python file:

1. Define a function **`coord(step_size)`** that returns **`+step_size`** 50% of the time, and **`–step_size`** 50% of the time.

In your notebook file (i.e. lab writeup):

1. Import your **`coord`** function in your notebook.
2. Create a variable **`step_size`** that sets the size of the step. Set it initially to 3.
3. Create a variable **`n_timesteps`** that sets the number of time steps that you will take in this random walk. Set this to 100 initially.
4. Create two arrays: one to store **`n_timesteps`** *x* positions and the other **`n_timesteps`** *y* positions.
5. Create a loop that will set the *x* and *y* arrays to store their respective positions at each time step. Inside this loop, use an i*f..else* logic statement to
    a) set the initial position to be at the origin if we are at the first time step and
    b) set all the subsequent *x* and *y* positions to be random steps (in *x* and *y*) of size **`step_size`** from the previous position (calling your **`coord(step_size)`** function would be helpful here).
6. Once you have created the arrays, print the final *x* and *y* positions to the screen. Confirm different runs of the program produce different final positions.

Part 2: Exploring the Program using the Tools in *Spyder*

Answer the following questions in comments in your program:

1. Use the Variable explorer feature of the notebook (by clicking on the "Variable explorer" tab in the upper righthand subwindow) to explore the values of the variables in your program after it has run. How can you confirm you set the initial position to the origin correctly?
2. Print the y position was after 50 time steps (be careful, this is NOT **`y[50]`** if your *y* array variable was named **`y`**). Explain how you determined this.
3. Change the value of the *y* position after 50 time steps. Explain how you did this.

Part 3: Visualizing the Random Walk

It would be nice to visualize this random walk as a line with a dot at the end.

1. Using the plotting command, plot the x and y position arrays to show the path your random walk took, using a line for the entire path and plotting the last point as a dot.
2. It would be nice to set up the plot to be centered at the origin, so determine the maximum x and y extent to use and set the **`xlim`** and **`ylim`** appropriately. **HINT:** You can use a combination of the **`max`** and **`absolute`** commands to determine the farthest your random walk gets from the origin (in each dimension). Set the limits to run from negative that value to positive that value.
3. Once you have plotting working, increase the number of time steps to 10000 and see what happens.

Optional Fun

A possibility for extra credit exists by doing any or all of the following:

1. Add a second random walker, plotting them on the same plot with a different color path and final point. Set the limits on the plot taking into account both walkers.
2. Figure out how to stop the execution of the program immediately after you create the empty arrays and comment on the values stored in those arrays right after they are created. Describe the method you used to stop the execution in your comments as well as your answer to this question.

What to turn in at the end of lab:

- Upload your code (.py file and .ipynb file) to your github repository for Lab 01 (link was sent via email).

Commands and structures you may need:

`import numpy as np`
`from random import random`
`import matplotlib.pyplot as plt`
	These will import commands from various Python packages.

`def function(x):` `  return y`	This allows one to define a function that takes an input of *x* and returns a value *y*
`random()`	This function from **`random`** package will generate a random number between 0 and 1.
`np.empty(N,float)`	Using the **`empty`** function imported from the **`numpy`** package, this creates an array of **`N`** floating point numbers.
`for n in range(N):`	Starts a **`for`**-loop that will change the variable **`n`** from **`0`** up to **`N-1`**

`if condition1 :`
`  statement`
`else:`
`  other statement`
	If *condition1* is true, then do the statement otherwise do the "other statement.

`np.absolute(x)`	Returns an array containing the absolute values of the **`x`** array.
`np.max([a,b,c])`	Returns the maximum value from the list of values **`a`**, **`b`**, and **`c`**. (can be any number of values).

`plt.plot(x, y, color='red', linestyle='solid')`
`plt.plot(x[-1], y[-1], color='red',`
`marker='o')`
	Plot a red line of all the values in **`x`** and **`y`**, treating them as pairs of x,y positions. The second line plots the last point as a red dot.

`plt.xlim(-5, 5)` `plt.ylim(-2, 9)`	Set the x and y plot limits.
`plt.show()`	You may need this to actually display the plot.