Carleton University
Department of Systems and Computer Engineering
SYSC 1005 - Introduction to Software Development - Fall 2013

Lab 9 - Using Lists and Tuples

Attendance/Demo

To receive credit for this lab, you must make an effort to complete the exercises, and demonstrate your work.

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. Finish any exercises that you don't complete by the end of the lab on your own time.

Getting Started

1. Launch Wing IDE 101.
2. In the editor, open a new file. Save this file as lab9.py.

Part 1 - Comparing Lists and Tuples

Exercise 1

Suppose we want to represent points on a two-dimensional Cartesian plane. We could store the (x, y) coordinates of each point in a list. Type the following statement, which represents the point (1.0, 2.0) as a list:

```
>>> point1 = [1.0, 2.0]
```

What is displayed when Python evaluates `point1`? Try this:

```
>>> point1
```

The problem with this approach is that Python lists are *mutable*. We could call the `append` method to insert a `float` at the end of the list. Try this:

```
>>> point1.append(3.0)
>>> point1
```

The list now represents a point with three coordinates, so we've changed it from a point in two dimensions to one in three dimensions.

We could then call the `pop` method on this list to remove an element. Try this:

```
>>> point1.pop(0) # Remove the list element at index 0
>>> point1
```

```
>>> point1.pop()   # Remove the last element in the list
>>> point1
```

The point represented by the list now has only one coordinate.

To avoid these issues, we should represent points using an *immutable* container.

Type these statements in the shell (note that the numbers are enclosed in parenthesis, not square brackets):

```
>>> point1 = (1.0, 2.0)
>>> point1
>>> type(point1)
```

What is displayed when variable `point1` is evaluated? What is the type of the object bound to `point1`?

Recall that a *tuple* is a container that is similar to a list, except that it can't be modified after is initialized. In the previous experiment, to represent the point (1.0, 2.0) we created a tuple containing two real numbers, 1.0 and 2.0.

As with lists, the objects stored in a tuple can be retrieved by using the `[]` operator to specify their position in the tuple. Type these statements to retrieve the *x* and *y* coordinates of the point represented by `point1`. What values are displayed when variables `x` and `y` are evaluated?

```
>>> x = point1[0]
>>> y = point1[1]
>>> x
>>> y
```

We can unpack all the objects in a tuple, binding them to individual variables, by using a statement of the form:

```
var_1, var_2, var_3, ..., var_n = t
```

where `t` is variable bound to a tuple containing *n* objects. This is equivalent to:

```
var_1 = t[0]
var_2 = t[1]
    ...
var_n = t[n-1]
```

Try this experiment. What values will be displayed when variables `x` and `y` are evaluated?

```
>>> point2 = (4.0, 6.0)
>>> x, y = point2
>>> x
>>> y
```

We can easily demonstrate that tuples are immutable. You can't replace objects in a tuple, or add objects to or remove objects from a tuple. Type these statements in the shell. What is displayed when each statement is executed?

```
>>> point2[0] = 2.0  # Change the point to (2.0, 6.0)?
>>> point2.append(4.0)
>>> point2.pop(0)
```

Exercise 2

In lab9.py, define a function that is passed two tuples, each representing a point on a two-dimensional plane. The function returns the distance between the two points. The function header is:

```
def distance(pt1, pt2):
```

Use the shell to test your function. Here is one test case:

```
>>> point1 = (1.0, 2.0)
>>> point2 = (4.0, 6.0)
>>> distance(point1, point2)
5.0
```

Exercise 3

Some graphics packages use the term *polyline* to refer to a line made up of multiple line segments. We can represent a polyline using a list of tuples.

For example, the following statement creates a list that represents the polyline defined by the points (1.0, 2.0), (4.0, 6.0) and (10.0, -2.0). Type these statements in the shell. What is displayed when `poly` is evaluated?

```
>>> poly = [(1.0, 2.0), (4.0, 6.0), (10.0, -2.0)]
>>> poly
```

We could also initialize the line this way. Try it:

```
>>> point1 = (1.0, 2.0)
>>> point2 = (4.0, 6.0)
>>> point3 = (10.0, -2.0)
>>> poly = [point1, point2, point3]
>>> poly
```

Or, we could call the append method to initialize the line, one point at a time. What is displayed when `poly2` is evaluated?

```
>>> poly2 = []
>>> poly2.append(point1)
>>> poly2.append(point2)
>>> poly2.append(point3)
```

```
>>> poly2
```

We can use a `for` loop to iterate over all the tuples in the list. What is displayed when this loop is executed?

```
>>> for point in poly:
...     point           # evaluate poly[0], poly[1], poly[2]
...
```

Predict whether or not the following loop will display the same values as the previous one. Execute the loop in the shell.

```
>>> for i in range(len(poly2)):
...     poly2[i]
...
```

Was your prediction correct?

Exercise 4

In lab9.py, define a function that is passed a list of tuples, with each tuple representing one point in a polyline. The function returns the sum of the lengths of all the line segments in the polyline. The function header is:

```
def sum_lengths(points):
```

Use the shell to test your function. Here is one test case:

```
>>> poly = [(1.0, 2.0), (4.0, 6.0), (10.0, -2.0)]
>>> sum_lengths(poly)
15.0
```

Part 2 - Curve Fitting Using the Method of Least Squares

Every engineering student has tackled the problem of fitting a line through a set of points obtained during a lab experiment.

Linear regression is a technique for fitting a curve through a set of points by applying a goodness-of-fit criterion. The most common form of linear regression is *least-squares fitting*. The mathematical derivation of this technique is beyond the scope of this course, but if you're interested, you can read this page: http://mathworld.wolfram.com/LeastSquaresFitting.html.

Suppose we have a set of n points, $\{ (x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1}) \}$. The equation of a straight line through these points has the form $y = mx + b$, where m is the slope of the line and b is the y-intercept.

Using the method of least squares, the slope and y-intercept of the line with the best fit are calculated this way:

$m = (sumx \times sumy - n \times sumxy) \div (sumx \times sumx - n \times sumxx)$

$b = (sumx \times sumxy - sumxx \times sumy) \div (sumx \times sumx - n \times sumxx)$

where:

sumx is $x_0 + x_1 + x_2 + \ldots + x_{n-1}$; i.e., the sum of all the x values

sumy is $y_0 + y_1 + y_2 + \ldots + y_{n-1}$; i.e., the sum of all the y values

sumxx is $x_0^2 + x_1^2 + x_2^2 + \ldots + x_{n-1}^2$; i.e., the sum of the all the squares of the x values

sumxy is $x_0 \times y_0 + x_1 \times y_1 + x_2 \times y_2 + \ldots + x_{n-1} \times y_{n-1}$; i.e., the sum of all the products of the (x, y) pairs

Exercise 5

To ensure that you understand these formulas, use the method of least squares to calculate the slope and y-intercept of the line through this set of points: {(1.0, 5.0), (2.0, 8.0), (3.5, 12.5)}. Don't write a program to do this; use a calculator. If your calculations are correct, the equation of the line will be:

$y = 3.0x + 2.0$.

Exercise 6

Download linear_regression.py from cuLearn and open the file in Wing IDE 101. This file contains a function named `get_points`, which returns a list of tuples. Each tuple represents one (x, y) point. In the shell, type these statements to verify that the list returned by this function contains three tuples:

```
>>> samples = get_points()
>>> len(samples)
>>> samples[0]
>>> samples[1]
>>> samples[2]
```

Exercise 7

In linear_regression.py, define a function named `fit_line_to_points` which is passed a list of tuples, with each tuple representing an (x, y) point. This function should use the method of least squares to calculate the slope and y-intercept of the best-fit straight line through the points. The slope and intercept must be returned in a tuple.

The function header is:

```
def fit_line_to_points(points):
```

Interactively test your function, passing it the list returned by `get_points`. Verify that the slope and

y-intercept returned by the function are 3.0 and 2.0.

Exercise 8

The block after the statement, `if __name__ == "__main__":` contains a single statement, `pass`. Replace `pass` with a short script that:

- calls `fit_line_to_points`, passing it the list of points returned by `get_points`;
- prints "The best-fit line is y = *m*x + *b*", where *m* and *b* are the values returned by your function.

Test your script.

Wrap-up

1. Remember to have a TA review your linear_regression.py file and give you a grade (Satisfactory, Marginal or Unsatisfactory) before you leave the lab.
2. Remember to save a copy of your linear_regression.py file (copy it to a flash drive, or email a copy to yourself, or store it on your M: drive - remember, files left on your desktop or in your Documents folder are not guaranteed to be there the next time you log in).
3. Log in to cuLearn, click the Submit Lab 9 link and submit linear_regression.py. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your files is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. After you've finished uploading the file, remember to click the Submit assignment button to change the submission status to "Submitted for grading".

Challenge Exercise

Suppose we want to fit lines through sets of points other than the ones returned by `get_points`. You could modify the script you wrote in Exercise 8 to read the (*x*, *y*) points from the keyboard, but this would be tedious and error-prone for sets of data with many points. Instead, we'll store the points in a text file that can be prepared with any text editor, and modify the script to read the points from the file.

Reading Strings from Text Files

Step 1: You've been provided with a file named data.txt that contains three (*x*, *y*) points, one per line:

1.0 5.0

2.0 8.0

3.5 12.5

Download this file from cuLearn to the same folder where linear_regression.py is stored.

To read data from a file, we must first open it, using Python's built-in `open` function. Type this:

```
>>> infile = open("data.txt", "r")
```

`open` takes two arguments, The first argument is a string containing the name of the file to open. The second argument is a string that specifies the mode; `"r"` means open the file for reading.

`open` returns an object that stores information about the opened file. Here, we've bound that object to variable `infile`.

Next, we'll use the `readline` method to read the file, one line at a time. Type these statements. What value is displayed each time variable `s` is evaluated?

```
>>> s = infile.readline()
>>> s
>>> s = infile.readline()
>>> s
>>> s = infile.readline()
>>> s
>>> s = infile.readline()
>>> s
>>> infile.close()
```

After you've finished reading a file, you should always close it by calling the `close` method.

Notice that `readline` returns each line as a string (an object of type `str`). The `readline` method returns an empty string (`''`) if it is called after all the lines have been read from the file.

Step 2: We can use a `for` loop to read lines from a file. Define the following function in linear_regression.py:

```
def read_and_print_lines():
    infile = open("data.txt", "r")
    for line in infile:
        print line
    infile.close()
```

On each iteration of the `for` loop, Python automatically calls `readline` and binds the line read from the file to variable `line`.

Call `read_and_print_lines` from the shell, and observe what is printed.

Converting Strings to Real Numbers

Step 3: Recall that the string representation of a real number can be converted to a `float` by calling Python's built-in `float` function. Try this:

```
>>> s = "2.0"
>>> x = float(s)
```

```
>>> x
```

Each line read from data.txt contains two numbers, but the `float` function doesn't work with strings containing multiple numbers. Try this:

```
>>> float('1.0   5.0')
```

We need to break up each line read from the file into two strings, each containing one number, then individually convert each string to a `float`. This is easy to do, using the `split` method. Try this:

```
>>> s = '1.0   5.0'
>>> numbers = s.split()
>>> numbers
```

Notice that `split` chops the string into two substrings, each containing one of the numbers, and returns a list containing both substrings.

How would you convert both strings in list `numbers` to values of type `float`? Design some experiments that show how to do this, and execute them in the shell.

Reading Points from a Text File

Step 4: Apply what you learned in Steps 1, 2 and 3 by defining a function named read_points in linear_regression.py. This function takes a single argument, a string containing the name of a text file. The function header is:

```
def read_points(filename):
```

Each line in the text file will contain two real numbers. The function will return a list of tuples, with each tuple containing one (x, y) point (i.e., a pair of `float`s).

Interactively test this function. If you call `read_points` with "data.txt" as the argument, the list returned by the function should be identical to the list returned by `get_points`.

Modifying the Curve-Fitting Script

Step 5: Modify your main script (not function `read_points`) so that it prompts the user to enter the name of a text file. The script should then read the points from the file and calculate the best-fit line through the points.

Test your script.