CIS 480 - Python - Fall 2005
WEEK 6 LAB EXERCISE and Homework #4

week 6 lab exercise due: Thursday, September 29th, END of lab
HW #4 due: Thursday, October 6th, 12:00 noon

Until I develop more formal opening comment block standards (which I plan to do), begin **each python module** that you write with at least the following opening comments:

* a comment containing the name of the module (the **file name** of the module, please --- **lab06.py**, for example)
* a comment containing your name, and
* a comment containing the date that your module was last modified

WEEK 6 LAB EXERCISE

For this lab exercise, you are **encouraged** to check your answers with classmates. In fact, you are **required** to do so with at least one classmate who has not yet had his/her work checked yet before you have me check your work; write the name(s) of those you checked your work with below:

____

(The point here is to have to argue, er, discuss with others why your answers are correct if they think they are not, or why others' are not correct if they think they are, but you do not...)

1. Answer the following questions in the space provided:

* Write an example of a Python empty dict literal: ____________________
* Write an example of a Python non-empty dict literal: ____________________

2. Assume that **price_dict** is a Python variable to which a dict literal has **already** been assigned, and that it currently contains at least 10 key-value pairs.

* write an expression whose value would be the number of key-value pairs in **price_dict**

  ____

* write an expression that would be the value corresponding to the key 'avocados' in **price_dict**

  ____

* write an expression that would be a list of all of the keys within **price_dict**

  ____

* write an expression that would be a list of all of the keys' values within **price_dict**

  ____

* write a statement that would **replace** the current value of the key 'apples' in **price_dict** with **1.99**

  ____

* write a statement that would **add** the new key 'grapes' with corresponding value 2.68 to **price_dict**.

____

* write an expression that would be true if 'cauliflower' is a key within **price_dict**.

____

* write a statement that would remove the key 'tea' and its value 4.99 from **price_dict**.

____

* Assume that **price_sum** is a variable that has been set to 0.0. Write a **for-loop** that, when finished, will have set **price_sum** to be the sum of the (assumed-to-be-numeric) keys' corresponding values in the dict **price_dict**.

3. A fun and useful function, pointed out to me by Eduardo Martinez: the **range** built-in function takes a beginning integer and an ending integer, and returns a list of the integers starting with the beginning integer and going up to but NOT including the ending integer. (How slice-like!)

So, `range(0, 6) == [0, 1, 2, 3, 4, 5]`

This function makes writing a so-called "count-controlled" for loop quite convenient: if you want to loop 10 times, this works nicely:

```
>>> for counter in range(0, 10):
...     print counter,
...
0 1 2 3 4 5 6 7 8 9
```

Write a statement that will create an empty dictionary **play_d**, followed by a for-loop that makes appropriate use of the **range** function to add to **play_d** the keys consisting of the integers 5 through 15 **inclusive**, each with a value of 0.

4. Imagine: one could certainly have a dictionary in which there were single-letter keys, each followed by a 2-element tuple representing the coordinates of a point in an x-y coordinate plane. We will assume that all of these x-y coordinates are non-negative (0 or higher).

```
my_points = { 'a':(4, 3), 'b':(1, 2), 'c':(5, 1) }
```

Our goal: to eventually get to the point in HW #4 below where we can make a simple ASCII display of an x-y coordinate plane with each of these key letters appearing at its value coordinates.

One useful little tool toward this goal would be a function **max_at_index**. It has two parameters: a dictionary that has the format described above, and an index into the value tuples. Its goal: to return the maximum value within all of the value tuples at that index.

For example,
```
max_at_index(my_points, 0) == 5
max_at_index(my_points, 1) == 3
```

Create a module **lab06.py**, and in it write the function **max_at_index**.

Then create a module **lab06_test.py**. In this module:

* import the **lab06.py** module,
* write print statement(s) that give the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "lab06.max_at_index({ 'a':(4, 3), 'b':(1, 2), 'c':(5, 1) }, 0)"
print "   should return 5: "
print lab06.max_at_index({ 'a':(4, 3), 'b':(1, 2), 'c':(5, 1) }, 0)
```

...for each of at least **TWO** different testing calls, one with an index of 0, and one with an index of 1, for an example dictionary with at least three appropriate key-value pairs.

When this runs like you would like, check your work with at least one classmate, and then write your name on the 'Next:' list on the board to get your work checked.

All of the above must be completed before the end of your lab time.

HOMEWORK #4

Create a Python module **hw04.py**. Within it, include the following:

0. Import the module **lab06.py**; we'll be using its **max_at_index** function later in this homework.

1. Write a function **ct_letter_freq**. **ct_letter_freq** expects one string parameter, expected to be a sentence, phrase, or other such language fragment. It returns a Python dictionary containing, for each letter within the passed string, a key consisting of the lowercase version of that letter and a value consisting of how many times that letter (uppercase OR lowercase) appears in the passed string. Note that letters that do not appear should not be keys in the resulting dictionary.

In a separate module **hw04_test.py**:

* import the **hw04.py** module,
* write a print statement that gives the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "hw04.ct_letter_freq('Apple of my Eye') should return:"
print "   (NOTE: key order WILL vary!!!)"
print "{'a':1, 'p':2, 'l':1, 'e':3, 'o':1, 'f':1, 'm':1, 'y':2 }"
print hw04.ct_letter_freq('Apple of my Eye')
```

...for each of at least **two** different testing calls:

* at least one with a some non-letter characters,
* at least one with a some uppercase letters,
* at least one with some letters appearing more than once,
* at least one that doesn't contain all of the letters in the alphabet.

(note that a single testing call could incorporate SEVERAL of the above...!)

2. Write a function **freq_bar_chart** that expects a dictionary of letter frequencies as its argument (such as the one returned by **ct_letter_freq**, conveniently enough!) It does not return anything, but it does print to the screen a simple ASCII horizontal bar chart of the letter frequencies that meets the following specifications:

* it should start with an appropriate, eye-catching title of your choice;

* then, each letter key, a dash, and then the number of X's corresponding to the value for that letter key should appear on its own line. Note that you may incorporate white space as you wish in this to make it look readable and attractive.

* the letter keys should appear in **alphabetical** order.

Add to module **hw04_test.py:**

* write print statement(s) that give the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "hw04.freq_bar_chart({'a':5, 'x':1, 'e':7})"
print "   should print a horizontal bar chart with:"
print "   a nice title, a followed by 5 X's, e followed by 7 X's,"
print "   and x followed by 1 X:"
hw04.freq_bar_chart({'a':5, 'x':1, 'e':7})
```

...for each of at least **two** different testing calls, each involving a dictionary with at least three key-value pairs.

3. Now, continuing with our goal of printing an ASCII x-y coordinate plane plotting the points in a dictionary containing single-letter keys and x-y coordinate tuples as the keys' values.

When you print this, the y coordinates will be vertical and in reverse order; the x coordinates will be from left to right.

It turns out that this is easier to accomplish if you create what we'll call a **plot** dictionary: this dictionary has keys that are the **y** coordinates, and for each y-coordinate key the value is a **list** of what should be printed "across" at that y-coordinate.

Write a function **build_blank_plot**. It takes a dictionary as described in lab exercise problem #4 and builds an "empty" plot dictionary of appropriate size for the points in the passed dictionary: it should meet the following specifications:

* its keys should be the integers from 0 to the maximum y coordinate found in the x-y coordinates in the keys' values in the passed dictionary;

* each key's value should be a list of strings of a single period, whose length is 1 plus the maximum x coordinate found in the x-y coordinates in the keys' values in the passed dictionary.

For example, for:

```
my_points = { 'a':(4, 3), 'b':(1, 2), 'c':(5, 1) }

hw04.build_blank_plot(my_points) ==
{3:['.', '.', '.', '.', '.', '.'],
 2:['.', '.', '.', '.', '.', '.'],
 1:['.', '.', '.', '.', '.', '.'],
 0:['.', '.', '.', '.', '.', '.']}
```

...although, of course, the resulting plot dictionary would not "print up" so nicely if you displayed its value to the screen (the keys would not be in reverse order, and you wouldn't get one list per line... 8-))

Add to module **hw04_test.py:**

* write print statement(s) that give the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "hw04.build_blank_plot({ 'a':(4, 3), 'b':(1, 2), 'c':(5, 1) })"
print "   should return a blank plot dictionary with:"
print "   * keys of 0 through 3, in any order;"
print "   * a list of 6 periods per key:"
print hw04.build_blank_plot({ 'a':(4, 3), 'b':(1, 2), 'c':(5, 1) })
```

...for each of at least **two** different testing calls, each involving a dictionary with at least three key-value pairs, at least one of which has DIFFERENT maximum x and maximum y coordinates amongst its value x-y coordinates.

4. Now, continuing in our goal to eventually print out the points in an ASCII x-y coordinate plane, write a function **add_points**. It expects a dictionary of single-letter keys with x-y coordinate tuples as its first argument (let's call this the point dictionary), and a plot dictionary such as that produced by build_blank_plot as its second argument.

It then returns a plot dictionary in which, for each single-letter key in the point dictionary, the plot dictionary has had the period in the y-coordinate key list at the x-coordinate idex into that list changed to that single letter key. Or, each single-letter key has been placed in the plot dictionary based on its x and y coordinates.

That is,

```
>>> hw04.add_points({'a':(4,3), 'b':(1,2), 'c':(5,1)},
...                 {3:['.','.','.','.','.','.'],
...                  2:['.','.','.','.','.','.'],
...                  1:['.','.','.','.','.','.'],
...                  0:['.','.','.','.','.','.']})
{0: ['.', '.', '.', '.', '.', '.'],
 1: ['.', '.', '.', '.', '.', 'c'],
 2: ['.', 'b', '.', '.', '.', '.'],
 3: ['.', '.', '.', '.', 'a', '.']}
```

...although, again, the resulting plot dictionary would not "print up" so nicely if you displayed its value to the screen (the keys would not be in order, and you wouldn't get one list per line... 8-))

Add to module **hw04_test.py:**

* write print statement(s) that give the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "hw04.add_points({'a':(4,3), 'b':(1,2), 'c':(5,1)},\n" +\
      "                 {3:['.','.','.','.','.','.'],\n" +\
      "                  2:['.','.','.','.','.','.'],\n" +\
      "                  1:['.','.','.','.','.','.'],\n" +\
      "                  0:['.','.','.','.','.','.']})"
print "should return a plot dictionary with:"
print "   a in y-list 3, at index 4 (5th position)"
print "   b in y-list 2, at index 1 (2nd position)"
print "   c in y-list 1, at index 5 (6th position)"
print hw04.add_points({'a':(4,3), 'b':(1,2), 'c':(5,1)},\
                 {3:['.','.','.','.','.','.'],\
                  2:['.','.','.','.','.','.'],\
                  1:['.','.','.','.','.','.'],\
                  0:['.','.','.','.','.','.']})
```

...for each of at least **two** different testing calls, each involving a point dictionary with at least three key-value pairs.

5. Whew! Now, write a function **display_plot**: it expects a plot dictionary as its parameter, and prints it to the screen, meeting the following specifications:

* display the y-coordinate scale on the left-hand side, vertically, in reverse order;
* display the x-coordinate scale along the bottom, in order;
* uses white space "nicely" to result in an attractive, readable display,
* prints each value --- either period or letter --- in the y-coordinate lists at its proper location.

(hint: remember that a print whose argument ends with a comma adds a space, but doesn't go to the next line yet...)

That is,

```
>>> hw04.display_plot({0: ['.', '.', '.', '.', '.', '.'],
...                    1: ['.', '.', '.', '.', '.', 'c'],
...                    2: ['.', 'b', '.', '.', '.', '.'],
...                    3: ['.', '.', '.', '.', 'a', '.']})

3 . . . . a .
2 . b . . . .
1 . . . . . c
0 . . . . . .
  0 1 2 3 4 5
```

Add to module **hw04_test.py:**

* write print statement(s) that give the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "hw04.display_plot({0: ['.', '.', '.', '.', '.', '.'],\n" +\
      "                  1: ['.', '.', '.', '.', '.', 'c'],\n" +\
      "                  2: ['.', 'b', '.', '.', '.', '.'],\n" +\
      "                  3: ['.', '.', '.', '.', 'a', '.']})"
print "should print an x-y coordinate plane with:"
print "   a at (4,3),"
print "   b at (1,2)"
print "   c at (5,1)"
hw04.display_plot({0: ['.', '.', '.', '.', '.', '.'],
                   1: ['.', '.', '.', '.', '.', 'c'],
                   2: ['.', 'b', '.', '.', '.', '.'],
                   3: ['.', '.', '.', '.', 'a', '.']})
```

...for each of at least **two** different testing calls, each involving a plot dictionary with at least three non-period points within it.

6. Let's add a final level, to conclude this (for this homework, anyway). Write a function **display_points** which takes as its argument a point dictionary, and displays to the screen an x-y coordinate plane with its keys appearing at its corresponding coordinates.

It must, of course, use **build_blank_plot**, **add_points**, and **plot_dict** appropriately.

Add to module **hw04_test.py:**

* write print statement(s) that give the actual call you are about to make and what it SHOULD return, before a print statement printing the actual result of that call --- for example,

```
print "hw04.display_points({'a':(4,3), 'b':(1,2), 'c':(5,1)})"
print "should print an x-y coordinate plane with:"
print "   a at (4,3),"
print "   b at (1,2)"
print "   c at (5,1)"
hw04.display_points({'a':(4,3), 'b':(1,2), 'c':(5,1)})
```

...for each of at least **two** different testing calls, each involving a point dictionary with at least three points within it, and with different maximum x and maximum y coordinates in each.

And, when you are satisfied, you should create a submittable output file **hw04_test.out**:

```
python hw04_test.py > hw04_test.out
```

By the due date and time given at the beginning of this handout, use **~st10/480submit** to submit your final versions of **hw04.py**, **hw04_test.py**, and **hw04_test.out** (And remember --- you can submit more than one version of these before the deadline, if inspiration strikes after a submission. As the syllabus notes, I'll simply grade the latest version that was submitted before the deadline.)