New York University
School of Continuing and Professional Studies
Division of Programs in Information Technology

Introduction to Python
Exercises, Session 5

Ex. 5.1 Given the following dict:

```
mydict = {'a': 1, 'b': 2, 'c': 3}
```

In two statements, add two key/value pairs to mydict (continuing the series values indicated). Print the dict.

Expected Output:

```
{'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4}
```

Ex. 5.2 Given the following list:

```
mylist = ['Hey', 'there', 'I', 'am', 'amazing!']
```

Initialize an empty dictionary. Loop through each element in mylist, add a key/value pair to the dict where the key is the word and the value is the len() of the word. Print the dict.

Expected Output:

```
{'I': 1, 'there': 5, 'am': 2, 'Hey': 3, 'amazing!': 8}
```

Ex. 5.3 Modify the above script: at the end of the script, instead of printing the dict directly, loop through the dict and print each key and value pair with a descriptive message (hint: use the str format() for easy embedding of values within a string).

Expected Output:

```
"I" is len 1.
"there" is len 5.
"am" is len 2.
"Hey" is len 3.
"amazing!" is len 8.
```

Ex. 5.4 Modify the above script: at the end of the script, instead of looping through the dict, use input() to ask for one of the words. If the user inputs one of the words, the program prints the word and its length.

Expected Output:

```
please enter a word:  there
the word "there" is len 5
```

Ex. 5.5 Extend the above script to allow for a missing key. If the user's input word does not exist in the dict, print a statement to that effect (hint: use the **in** operator to test for the existence of a key).

Sample program run(s):

```
$ python myprog.py
please enter a word:  I
the word "I" is len 1

$ python myprog.py
please enter a word:  Jamie
the word "Jamie" does not exist in the dictionary
```

Ex. 5.6 Start with an empty dict. Opening and looping through the file revenue.txt, load the name of the store as the key and the decimal point value as the value in the dict. Print the dict.

Expected Output:

```
{"Hipster's": '11.98\n', 'Dothraki Fashions': '5.98\n',
 "Awful's": '23.95\n', "Haddad's": '239.50\n',
 'The Clothiers': '115.20\n', 'The Store': '211.50\n',
 'Westfield': '53.90\n'}
```

Ex. 5.7 Fix the above script by removing the newline from each value (hint: remove it when it is still part of the line by calling rstrip() on the line, before splitting.

Expected Output:

```
{"Hipster's": '11.98', 'Dothraki Fashions': '5.98',
 "Awful's": '23.95', "Haddad's": '239.50',
 'The Clothiers': '115.20', 'The Store': '211.50',
 'Westfield': '53.90'}
```

Ex. 5.8 Given the following string:

```
personal_info = "595-33-9193:68:Columbus, OH"
```

Split the data into a list of strings. Then in a single statement insert it into a string so that it appears as shown in the output. (Hint: use a triple-quoted multiline string, i.e.

```
print("""SS#: {}
Age: {}
Residence: {}""".format())
```

or a single string with newlines (\n), i.e.

```
print("SS#: {}\nAge: {}\nResidence: {}".format())
```

between each line, where the call to **format()** will contain as arguments the three values taken from the split.

Expected Output:

```
SS#:  595-33-9193
Age:  68
Residence:  Columbus, OH
```

Ex. 5.9 Starting with the following list:

```
cities = ['Boston', 'Chicago', 'New York', 'Boston', 'Chicago', 'Boston']
```

Count the occurrence of each value in the list by compiling a count of cities in the dict, by storing the city as a key in the dict and an integer count of the number of times that city occurred. However, don't attempt to loop through the list more than once. Hint: as your loops encounters each element in the list, use the **in** membership test to see if the value is a key in the dict. If it isn't, add it as a key with a value of 0. Then, still in the block, and whether or not it is in the dict, add one to the value for that key, and assign the new +1 value back to the dict for that key.

Expected Output:

```
{'Boston': 3, 'Chicago': 2, 'New York': 1}
```

Ex. 5.10 Start with an empty dictionary. Opening and reading student_db.txt (make sure to account for the header line by slicing readlines()), build a dictionary of states and a count for each state. Print the dictionary. Make sure to use only the dictionary to count -- no independing counting with an integer, list, etc.

Expected Output:

```
{'NY': 4, 'NJ': 2, 'PA': 1}
```

Hint: the key to making a counting dictionary work is in checking to see if any given key ito be counted is new to the dictionary. If the key is new, it must be added to the dictionary. If it isn't new, it will be rewritten to the dictionary, but with a new value that is (in this case) one more than the value it held earlier. So consider the dict equivalent of this:

```
x = x + 1                      # would only work if 'x' already exists
```

Here is the dict equivalent:

```
mydict['x'] = mydict['x'] + 1  # would only work if key 'x' already exists
```

where 'x' is a value that may appear multiple times in the data, and our object is to count it (as well as any other values that appear).

So, since we can't know whether any given key is new, we must check:

```
if 'x' not in mydict:
```

Basically the simplest way to make this work is check to see if the key is new; if it is, set it in the dict with a value of 0. Then, whether or not it is new, you can go ahead and add 1 to it - if it is new, the new value with this key will be 1. If it is not new, this adding will simply increment the value.

Ex. 5.11 Now reading from revenue.txt, sum up the values associated with each state. (Hint: this will be very much like the counting dictionary, but we will be adding float values to a sum (that starts at 0.0 for new keys; see the above explanation for working with keys new to the dictionary vs. "old" keys).)

Expected Output:

```
{'NY': 133.16, 'NJ': 265.4, 'PA': 263.45}
```

Ex. 5.12 Starting with the following dictionary:

```
mydict = {'c': 0.3, 'b': 7, 'a': 5}
```

Sort this dictionary by letter. Since the sorted() function will return a list of dictionary keys, you can then loop through those keys to print the sorted keys, and use each key with the dictionary to look up each value associated with it. Loop through the sorted keys, look up each value, and output each pair on a line.

Expected Output:

```
a => 5
b => 7
c => 0.3
```

Ex. 5.13 Modifying the previous solution, sort the dictionary by value. This done using the **key=mydict.get** argument to sorted() (assuming the variable name of your dictionary is mydict).

Expected Output:

```
c => 0.3
a => 5
b => 7
```

Ex. 5.14 Modifying the previous solution, reverse the sort using the reverse=True argument to sorted().

Expected Output:

```
b => 7
a => 5
c => 0.3
```

Ex. 5.15 Modifying the previous solution, allow the user to enter an argument indicating which direction the sort should go (the user can enter "ascending" or "descending"). Then using an if/else and testing the user's input, if "ascending", set a new variable rev to False; if "descending", set variable rev to True. Finally, use this variable rev as part of the reverse argument, i.e.:

```
sorted_keys = sorted(mydict, key=mydict.get, reverse=rev)
```

Sample program run(s):

```
$ python myprog.py
please enter a direction:  ascending
c => 0.3
a => 5
b => 7

$ python myprog.py
please enter a direction:  descending
b => 7
a => 5
c => 0.3

$ python myprog.py
please enter a direction:  arskando
enter "ascending" or "descending"
```

Ex. 5.16 Extending the summing dictionary (or the counting dictionary) taken from revenue.txt, once the loop is complete, sort the dictionary using a standard sort (which will sort the dict by state names).

Expected Output:

```
# (note that the states are sorted alphabetically):
NJ => 265.4
NY => 133.16
PA => 263.45
```

Ex. 5.17 Now sort the summing dictionary by value.

Expected Output:

```
NY => 133.16
PA => 263.45
NJ => 265.4
```