CptS355 - Assignment 3 (Python)
Spring 2018

Python Warm-up

Assigned: Friday, March 2nd, 2018

Due: Monday, March 12th, 2018

Weight: This assignment will count for 6% of your final grade.

This assignment is to be your own work. Refer to the course academic integrity statement in the syllabus.

Turning in your assignment

All the problem solutions should be placed in a single file named **HW3.py**.When you are done and certain that everything is working correctly, turn in your file by uploading on the Assignment-3(Python) DROPBOX on Blackboard (under AssignmentSubmisions menu). The file that you upload must be named **HW3.py**. Be sure to include your name as a comment at the top of the file. Also in a comment, indicating whether your code is intended for Unix/Linux or Windows –programs' behavior may differ slightly between the two systems. You may turn in your assignment up to 4 times. Only the last one submitted will be graded. Implement your code for Python3.

The work you turn in is to be your own personal work. You may NOT copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself.

Grading

The assignment will be marked for good programming style (appropriate algorithms, good indentation and appropriate comments -- refer to the Python style guide) -- as well as thoroughness of testing and clean and correct execution. You will lose points if you don't (1) provide test functions / additional test cases, (2) explain your code with appropriate comments, and (3) follow a good programming style. **For each problem below, around 20% of the points will be reserved for the test functions and the programming style.**

- Good python style favors for loops rather than while loops (when possible).
- Also, code to do the same thing (or something easily parameterizable) at different points in a program should not be duplicated but extracted into a callable function.
- Turning in "final" code that produces debugging output is bad form, and points may be deducted if you have extensive debugging output. We suggest you the following:
    - Near the top of your program write a debug function that can be turned on and off by changing a single variable. For example,

      ```
      debugging = True
      def debug(*s): if debugging: print(*s)
      ```

        - Where you want to produce debugging output use:
          `debug(”This is my debugging output”)`
          instead of `print`.
        - (How it works: Using * in front of the parameter of a function means that a variable number of arguments can be passed to that parameter. Then using *s as print's argument passes along those arguments to print.)

Problems:

1. **(Dictionaries)**

a) **`addDict(d)` – 10%**

Assume you keep track of the number of hours you study every day for each of the courses you are enrolled in. You maintain the weekly log of your hours in a Python dictionary as follows:

```
{'Mon':{'355':2,'451':1,'360':2},'Tue':{'451':2,'360':3},
'Thu':{'355':3,'451':2,'360':3}, 'Fri':{'355':2},
'Sun':{'355':1,'451':3,'360':1}}
```

The keys of the dictionary are the days you studied and the values are the dictionaries which include the number of hours for each of the courses you studied. Please note that you may not study for some courses on some days OR you may not study at all on some days of the week.

Define a function, **`addDict(d)`** which adds up the number of hours you studied for each of the courses during the week and returns the summed values as a dictionary. Note that the keys in the resulting dictionary should be the course names and the values should be the total number of hours you have studied for the corresponding courses. **`addDict`** would return the following for the above dictionary:

```
{'355': 8, '451': 8, '360': 9}
```

(*Important note*: Your function should not hardcode the course numbers and days of the week. It should simply iterate over the keys that appear in the given dictionary and should work on any dictionary with arbitrary course numbers and days of the week)

(Important note: When we say a function returns a value, it doesn't mean that it prints the value. Please pay attention to the difference.)

Define a function `testaddDict()` that tests your `addDict(d)` function, returning `True` if the code passes your tests, and `False` if the tests fail. You can start with the following code:

```
def addDict(d):
   #write your code here

def testaddDict():
   #write your code here
```

b) **`addDictN(L)` – 10%**

Now assume that you kept the log of number of hours you studied for each week on each class throughout the semester and stored that data as a list of dictionaries. This list includes a dictionary for each week you recorded your log. Assuming you kept the log for N weeks, your list will include N dictionaries.

Define a function `addDictN` which takes a list of weekly log dictionaries and returns a dictionary which includes the total number of hours that you have studied for your enrolled courses throughout the semester. Your function definition should use the Python map and reduce functions as well as the `addDict` function you defined in part(a). You may need to define an additional helper function.

Example:

Assume you have recorded your log for 3 weeks only.

```
[{'Mon':{'355':2,'360':2},'Tue':{'451':2,'360':3},'Thu':{'360':3},
'Fri':{'355':2}, 'Sun':{'355':1}},
{'Tue':{'360':2},'Wed':{'355':2},'Fri':{'360':3, '355':1}},
{'Mon':{'360':5},'Wed':{'451':4},'Thu':{'355':3},'Fri':{'360':6},
'Sun':{'355':5}}]
```

For the above dictionary `addDictN` will return:

```
{'355': 16, '360': 24, '451': 6}
```

(The items in the dictionary can have arbitrary order.)

2. Dictionary and List Comprehensions

(a) `charCount(s)` – 5%
Define a function, `charCount(s)` counting the number of times that each character appears in a given string `s`. `charCount(s)` should return a list of characters in the input string `s` each paired with its frequency (the number of times character appears in the string). The "space character" should be excluded in the result.

Characters must appear in the list ordered from least frequent to most frequent. For example,
`charCount('Cpts355 --- Assign1')` is:

```
[('1', 1), ('3', 1), ('A', 1), ('C', 1), ('g', 1), ('i', 1), ('n', 1),
('p', 1), ('t', 1), ('5', 2), ('-', 3), ('s', 3)]
```

Characters with the same frequency must appear in increasing alphabetical order. To implement sorting you can use the Python built-in function `sorted`. You don't need to write your own sorting code.
(Hints: Use a dictionary to maintain the counts. But remember that your function should return a list of tuples. You should be able to build the dictionary in a single pass over the input string.)

You can start with the following code:

```
def charCount(s):
      #write your code here
```

(b) `charCount2(s)` – 5%
Rewrite your `charCount` function using list comprehension. Name this function `charCount2(s)`.
(Hint: Use the `count()` method of string to get the number of occurrences of a character in a string. For example: `“how many a’s are there”.count('a')` returns 3)

Define a function `testcharCount()` that tests your `charCount(s)` and `charCount2(s)`functions, returning `True` if the code passes your tests, and `False` if the tests fail.

You can start with the following code:

```
def charCount2(s):
      #write your code here

def testcharCount():
      #write your code here; see the sample test function on page#4
```

3. List and Dictionary

a) `lookupVal(L,k)`– 5%
Write a function `lookupVal` that takes a list of dictionaries `L` and a key `k` as input and checks each dictionary in `L` starting from the end of the list. If `k` appears in a dictionary, `lookupVal` returns the value for key `k`. If `k` appears in more than one dictionary, it will return the one that it finds first (closer to the end of the list).

For example:

```
L1 = [{"x":1,"y":True,"z":"found"},{"x":2},{"y":False}]
```

```
lookupVal(L1,"x") returns 2
lookupVal(L1,"y") returns False
lookupVal(L1,"z") returns "found"
lookupVal(L1,"t") returns None
```

b) `lookupVal2(tL,k)` – 10%

Write a function `lookupVal2` that takes a list of tuples (`tL`) and a key `k` as input. Each tuple in the input list includes an integer index value and a dictionary. The index in each tuple represent a link to another tuple in the list (e.g. index 3 refers to the 4th tuple, i.e., the tuple at index 3 in the list) `lookupVal2` checks the dictionary in each tuple in `tL` starting from the end of the list and following the indexes specified in the tuples.

For example, assume the following:

```
[(0,d0),(0,d1),(0,d2),(1,d3),(2,d4),(3,d5),(5,d6)]
   0       1      2      3      4      5      6
```

The `lookupVal2` function will check the dictionaries `d6,d5,d3,d1,d0` in order (it will skip over `d4` and `d2`) The tuple in the beginning of the list will always have index 0.

It will return the first value found for key `k`. If `k` is couldn't be found in any dictionary, then it will return `None`.

For example:

```
L2 = [(0,{"x":0,"y":True,"z":"zero"}),
      (0,{"x":1}),
      (1,{"y":False}),
      (1,{"x":3, "z":"three"}),
      (2,{})]
```

```
lookupVal2(L2,"x") returns 1
lookupVal2(L2,"y") returns False
lookupVal2(L2,"z") returns "zero"
lookupVal2(L2,"t") returns None
```

(*Note*: I suggest you to provide a recursive solution to this problem.
Hint: Define a helper function with an additional parameter that hold the list index which will be searched in the next recursive call.)

4. (Higher Order Functions) `funRun(d,name,args)` – 10%

Write a function `funRun(d,name,args)` that takes a dictionary (`d`), a function name (`name`), and the list of arguments that will be passed to the function `f (i.e.,args)`. In dictionary `d`, the keys are the function names and values are the functions themselves. `funRun` calls the function for key `name` with the provided arguments in `args` and returns whatever the function returns. If the number of arguments in `args` doesn't match the expected number of arguments, your function should return an error message (and should not break). (You may assume that functions in dictionary can take at most 3 arguments.)

Example:

```
d = {"add": lambda x,y: (x+y), "concat3": lambda a,b,c:
(a+","+b+","+c),"mod2": lambda n: (n % 2)}
```

```
funRun(d, "concat3", ["one","two","three"]
returns "one,two,three"
```

```
funRun(d, "mod2", [40])
returns 0
```

You can start with the following code:

```
def funRun(d, name, args):
      #write your code here

def testfunRun():
      #write your code here; see the sample test function on page#4
```

(*Note*: To find out the number arguments each function takes, use the inspect function of Python. Make sure to import the "inspect" module.

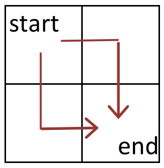
Example:

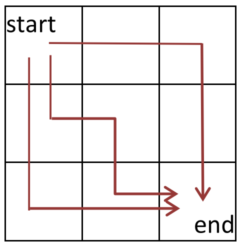
```
import inspect
def add(a,b):
   return a + b

inspect.getfullargspec(add)
#returns
FullArgSpec(args=['a', 'b'], varargs=None, varkw=None, defaults=None,
kwonlyargs=[], kwonlydefaults=None, annotations={})
#To get the number of arguments:
len(inspect.getfullargspec(add).args)
2
```

5. **(Recursion) `numPaths(m,n)` – 10%**

Consider a robot in a MxN grid who is only capable of moving right or down in the grid. The robot starts at the top left corner, (0,0), and is supposed to reach to the bottom right corner, (M-1,N-1). Write a function `numPaths` that takes the grid length and width (i.e., M, N) as argument and returns the number of different paths the robot can take from the start to the goal. Give and answer using recursion. (A correct solution without recursion will worth half the points.)





For example, the 2x2 grid has a total of two ways for the robot to move from the start to the goal. For the 3x3 grid, the robot has 6 different paths (only 3 are shown above).

```
numPaths(2,2) returns 2
numPaths(3,3) returns 6
```

You can start with the following code:

```
def numPaths(m,n):
      #write your code here

def testnumPaths():
      #write your code here; see the sample test function on page#4
```

6. **Iterators**

a) **`iterSquares()`– 10%**

Create an iterator that represents the sequence of squares of natural numbers starting at 1.
For example:

```
>>> squares = iterSquares()
>>> squares.__next__()
1
>>> squares.__next__()
4
>>> squares.__next__()
9
```

You can start with the following code:

```
class iterSquares():
    #write your code here
```

b) **`numbersToSum(iNumbers,sum)` – 10%**

Define a function `numbersToSum` that takes an iterator "`iNumbers`" (representing a sequence of positive integers) and a positive integer value `sum`, and returns the next `n` elements from `iNumbers` such that the next `n` elements of the iterator add to less than `sum`, but the next `(n + 1)` elements of the iterator add to `sum` or more. `numbersToSum` should return a list of integers. (*Note*: Your implementation of `numbersToSum` doesn't need to be recursive. You can't assume a minimum value for `sum`. Note that the iterator retrieves the next element in the sequence in the second call to `numbersToSum`.)

For example:

```
>>> squares = iterSquares()
>>> numbersToSum(squares,55)
[1, 4, 9, 16]
>>> numbersToSum(squares,100)
[25, 36]
```

You can start with the following code:

```
def numbersToSum(iNumbers,sum):
    #write your code here

def testnumbersToSum():
    #write your code here; see the sample test function on page#4
```

7. **Streams**

a) **`streamSquares(k)` – 5%**

Using the `Stream` class we defined in the lecture, write a function `streamSquares(k)` that creates an infinite stream of positive integers starting at `k` (i.e., make the first element of the stream `k`).
For example:

```
>>> sqStream = streamSquares(25)
>>> myList = []
>>> while sqStream.first < 225:
       myList.append(sqStream.first)
       sqStream =sqStream.rest
```

Value of the `myList` will be `[25, 36, 49, 64, 81, 100, 121, 144, 169, 196]`

You can start with the following code:

```
def streamSquares(k):
    #write your code here
```

b) evenStream(stream) – 10%

Write a function `evenStream` that takes a stream of integers as input and returns the Stream of even integers from the input stream.

For example:
```
>>> evenS = evenStream(streamSquares(9))
>>> myList = []
>>> while evenS.first < 225:
       myList.append(evenS.first)
       evenS =evenS.rest
```

Value of the `myList` will be `[16, 36, 64, 100, 144, 196]`

You can start with the following code:
```
def evenStream(stream):
    #write your code here

def testevenStream ():
    #write your code here; see the sample test function on page#4
```

Test your code:

Here is an example test function for testing `trans`. Make sure to include 3 test cases for each function.

```
# function to test charCount
# return True if successful, False if any test fails

def testcharCount():
    if charCount("abcdef abcde abcd abc ab ab") !=
        [('f', 1), ('e', 2), ('d', 3), ('c', 4), ('a', 6), ('b', 6)]:
        return False
    if charCount("Passing 355") !=
     [('3', 1), ('P', 1), ('a', 1),('g', 1), ('i', 1), ('n', 1), ('5', 2), ('s', 2)]:
        return False
    if charCount("WSU-WA") != [('-', 1), ('A', 1), ('S', 1), ('U', 1), ('W', 2)]:
        return False
    return True
```

Go on writing test code for ALL of your code here; think about edge cases, and other points where you are likely to make a mistake.

Main Program

So far we've just defined a bunch of functions in our HW3.py program. To actually execute the code, we need to write the code for the "main" program. Unlike in C or Java, this is not done by writing a function with a special name. Instead the following idiom is used. This code is to be written at the left margin of your input file (or at the same level as the `def` lines if you've indented those.

```
if __name__ == '__main__':
   ...code to do whatever you want done...
```

For this assignment, we want to run all the tests, so your main should look like:

```
if __name__ == '__main__':
    passedMsg = "%s passed"
    failedMsg = "%s failed"
    if testcharCount():
       print ( passedMsg % 'charCount' )
    else:
       print ( failedMsg % 'charCount' )
    # etc. for the other tests.
    # notice how you are repeating a lot of code here
    # think about how you could avoid that
```