CptS355 - Lab 3 (Python) - Fall 2021

Assigned: Friday, October 15, 2021

Weight: Lab3 will count for 1.5% of your course grade.

This assignment provides experience in Python programming.

Turning in your lab submission

This lab involves 6 programming problems and your solution will consist of a sequence of function definitions for the given problems. You should implement as many of the problems as you can. As explained in the "Grading" section, 4 correct solutions are sufficient to earn full points in the lab. Note that some problems have sub-parts; you should complete all parts to get credit for those problems.

You will write all your functions in the attached `Lab3.py` file. Attached file, `Lab3SampleTests.zip,` includes Python `unittest` tests for each problem. Make sure to test your code before you submit. Please refer to the "Testing your functions" section at the end of this document.

To submit your assignment, please upload the file `Lab3.py` on the Lab3 (Python) dropbox on Canvas (under Assignments). You may turn in your lab submission up to 3 times. Only the last one submitted will be graded.

Grading

The lab assignments will be auto-graded using automated tests. Your lab grade will be calculated as follows:

- When you submit the lab, you will automatically earn 60 points.
- For every correct solution that passes all test cases, you will earn 10 points. If your function fails even a single test case, you will not be able to earn points for that question.
- The lab includes 6 programming problems. You can implement as many of the lab problems as you want. However, 4 correct answers are sufficient to earn full points (100) in the lab. No extra credit will be given for additional solutions.
- However, submitting solutions for more than 4 problems will improve your chance to obtain 100% score (in case one of the test cases fail for the other problems).
- The Lab Zoom sessions are optional. You can work on the lab problems yourself or with your friends and submit your solutions on Canvas.

Problems:

1. **`getNumCases(data,counties,months)`**

   Assume that you work for a "Healthcare Data Analytics" company and you write scripts to process various dataset. In your analysis, you use the CDC's COVID-19 dataset. For example, the following dataset reports the monthly new COVID cases in year 2020 for some counties in WA.

   ```
   CDCdata =
   {'King':{'Mar':2706,'Apr':3620,'May':1860,'Jun':2157,'July':5014,'Aug':4327,'Sep':2843},
   'Pierce':{'Mar':460,'Apr':965,'May':522,'Jun':647,'July':2470,'Aug':1776,'Sep':1266},
   'Snohomish':{'Mar':1301,'Apr':1145,'May':532,'Jun':568,'July':1540,'Aug':1134,'Sep':811},
   'Spokane':{'Mar':147,'Apr':222,'May':233,'Jun':794,'July':2412,'Aug':1530,'Sep':1751},
   'Whitman' : {'Apr':7,'May':5,'Jun':19,'July':51,'Aug':514,'Sep':732, 'Oct':278} }
   ```

   The keys of the dictionary are the county names, and the values are the dictionaries which include the monthly new COVID cases. Note that some counties may not have any new cases in some months.
   Define a function, **`getNumCases`**, which calculates the total number of new cases for a given list of counties during a given list of months. For example:

   `getNumCases(CDCdata,['Whitman'],['Apr','May','Jun'])` returns 31, and

   `getNumCases(CDCdata,['King','Pierce'],['July','Aug'])` returns 13587.

   (*Important note*: Your function should not hardcode the county names and the month abbreviations. It should simply iterate over the keys that appear in the given dictionary. You will be deducted points if you hardcode any keys.)

   You can start with the following code:
   ```
   def getNumCases(data,counties,months):
      #write your code here
   ```

2. **`getMonthlyCases(data)`**

   Assume, your supervisor asks you to reformat the data and create a dictionary that includes the number of cases for each county, organized by months. For example, when you reformat the above dictionary you will get the following.

   ```
   {'Mar':{'King':2706,'Pierce':460,'Snohomish':1301,'Spokane':147},
   'Apr':{'King':3620,'Pierce':965,'Snohomish':1145,'Spokane':222,'Whitman':7},
   'May':{'King':1860,'Pierce':522,'Snohomish':532,'Spokane':233,'Whitman':5},
   'Jun':{'King':2157,'Pierce':647,'Snohomish':568,'Spokane':794,'Whitman':19},
   'July':{'King':5014,'Pierce':2470,'Snohomish':1540,'Spokane':2412,'Whitman':51},
   'Aug':{'King':4327,'Pierce':1776,'Snohomish':1134,'Spokane':1530,'Whitman':514},
   'Sep':{'King':2843,'Pierce':1266,'Snohomish':811,'Spokane':1751,'Whitman':732},
   'Oct':{'Whitman':278}}
   ```

   Define a function `getMonthlyCases` that reformats the CDC data as described above. Your function should not hardcode the county names and the month abbreviations.
   (The items in the output dictionary can have arbitrary order.)

3. **`mostCases(data)`**

Assume, you would like to find the month that had the maximum total number of new cases in all counties. For example:

```
mostCases(CDCdata) returns ('July', 11487)
#i.e., July has the max number of total new cases, which is 11487.
```

Your function definition should not use loops or recursion but use the Python `map` and `reduce` functions. You should also use the `getMonthlyCases` function you defined in part(b). You may define and call helper (or anonymous) functions, however your helper functions should not use loops or recursion.

4. **Dictionaries and Lists**

a) **`searchDicts(L,k)` – 5%**

Write a function `searchDicts` 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, `searchDicts` 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}]

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

You can start with the following code:

```
def searchDicts(L,k):
   #write your code here
```

b) **`searchDicts2(tL,k)` – 10%**

Write a function `searchDicts2` 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) `searchDicts2` 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 `searchDicts2` 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,{})]
```

```
searchDicts2 (L2,"x") returns 1
searchDicts2 (L2,"y") returns False
searchDicts2 (L2,"z")  returns "zero"
searchDicts2 (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.)
You can start with the following code:

```
def searchDicts2(L,k):
   #write your code here
```

5. `getLongest(L)` – 10%

Write a function, `getLongest`, which takes an arbitrarily nested list of strings (L) and it returns the longest string in L. Note that the longest string can be found at any nesting level, so your function should recursively check all sublists. You should not assume a max depth for the nesting. If there are more than one string that have the max length, you should return the one that appears earlier in the list.

For example:

```
getLongest(['1',['22',['333',['4444','55555',['666666']],'7777777'],'4444'],'22'])
returns '7777777'
```

```
getLongest([['cat',['dog','horse'],['bird',['bunny','fish']]]])
returns 'horse'
```

You can start with the following code:

```
def getLongest (L):
      #write your code here
```

6. Iterators

`apply2nextN()` – 20%

Create an iterator that represents the aggregated sequence of values from an input iterator. The iterator will be initialized with a combining function (op), an integer value (n) , and an input iterator (`input`). When the iterator's `__next__()` method is called, it will combine the next "n" values in the "`input`" by applying the "op" function and it will return the combined value. The iterator should stop when it reaches the end of the input sequence. If the input sequence is infinite, the `apply2nextN` will return an infinite sequence as well.
For example:

```
iSequence = apply2nextN(lambda a,b:a+b, 3, iter(range(1,32)))
# iSequence represents the sequence [6, 15, 24, 33, 42, 51, 60, 69, 78, 87, 31]

iSequence.__next__()   # returns 6
iSequence.__next__()   # returns 15
iSequence.__next__()   # returns 24
rest = []
```

```
for item in iSequence:
   rest.append(item)
# rest is [33, 42, 51, 60, 69, 78, 87, 31]

strIter =iter('aaaabbbbccccddddeeeeffffgggghhhhjjjjkkkkllllmmmm')
iSequence = apply2nextN(lambda a,b:a+b, 4, strIter)
iSequence.__next__()   # returns 'aaaa'
iSequence.__next__()   # returns 'bbbb'
iSequence.__next__()   # returns 'cccc'
rest = []
for item in iSequence:
    rest.append(item)
# rest is ['dddd','eeee','ffff','gggg','hhhh','jjjj','kkkk','llll','mmmm']
```

You can start with the following code:

```
class apply2nextN ():
    #write your code here
```

Testing your functions

We will be using the `unittest` Python testing framework in this assignment. See https://docs.python.org/3/library/unittest.html for additional documentation.

The file `Lab3SampleTests.py` provides some sample test cases comparing the actual output with the expected (correct) output for some problems. This file imports the `Lab3` module (`Lab3.py` file) which will include your implementations of the given problems.

In Python `unittest` framework, each test function has a "`test_`" prefix. To run all tests for problem 1, execute the following on the command line.

```
python -m unittest Q1_tests.py
```

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v Q1_tests.py
```

Repeat the above for other lab problems by changing the test file name, i.e. , `Q2_tests.py, Q3_tests.py, etc.`

In this assignment, we simply write some unit tests to verify and validate the functions. If you would like to execute the code, you 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...
```