

CptS355 - Assignment 3

Fall 2019

Python Warm-up

Assigned: Monday, October 7, 2019

Due: Thursday, October 17th, 2019

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. The file that you upload must be named **HW3.py**. You may turn in your assignment up to 3 times. Only the last one submitted will be graded. Implement your code for Python3.

At the top of the file in a comment, please include your name and the **names of the students with whom you discussed any of the problems in this homework**. This is an individual assignment and the final writing in the submitted file should be **solely yours**. 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 5% 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).
- 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:
 - o 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)
```
 - o Where you want to produce debugging output use:

```
debug("This is my debugging output",x,y)
```

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) `sprintLog(sprnt)` – 15%

Assume a software development team follows an agile process model. During each sprint, developers keep track of the number of hours they spent on each task during the sprint. They store a weekly log of their hours in a Python dictionary. For example:

```
{'John': {'task1': 5}, 'Rae': {'task1': 10, 'task2': 4}, 'Kelly': {'task1': 8, 'task3': 5}, 'Alex': {'task1': 11, 'task2': 2, 'task3': 1}, 'Aaron': {'task2': 15}, 'Ethan': {'task3': 12}, 'Helen': {'task3': 10}}
```

The keys of the dictionary are the names of the developers and the values are the dictionaries which include the task names and the number of hours spent for each task. Please note that each developer may work on several tasks during a sprint and multiple developers may work on the same tasks.

Define a function `sprintLog(sprnt)` which takes a dictionary of users as shown above and returns a dictionary of tasks, where each task is associated with the users who worked on that task during the sprint. For the above dictionary, `sprintLog` will return the following:

```
{'task1': {'John': 5, 'Rae': 10, 'Kelly': 8, 'Alex': 11}, 'task2': {'Rae': 4, 'Alex': 2, 'Aaron': 15}, 'task3': {'Kelly': 5, 'Alex': 1, 'Ethan': 12, 'Helen': 10}}
```

(Important note: Your function should not hardcode the developer and task names. It should simply iterate over the keys that appear in the given dictionary and should work on any dictionary with arbitrary names.)

(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.)

You can start with the following code:

```
def sprintLog (sprnt):  
    #write your code here
```

b) `addSprints(sprint1, sprint2)` – 10%

Now define a function, `addSprints(sprint1, sprint2)`, which takes the logs of two sprints as input and joins them. The result will be a joined dictionary including tasks from both dictionaries. The logs for the tasks that are common to both dictionaries will be merged. See below for an example.

```
sprint1 = {'task1': {'John': 5, 'Rae': 10, 'Kelly': 8, 'Alex': 11}, 'task2': {'Rae': 4, 'Alex': 2, 'Aaron': 15}, 'task3': {'Kelly': 5, 'Alex': 1, 'Ethan': 12, 'Helen': 10}}
```

```
sprint2 = {'task1': {'Mark': 5, 'Kelly': 10, 'Alex': 15}, 'task2': {'Mark': 2, 'Alex': 2, 'Rae': 10, 'Aaron': 10}, 'task4': {'Helen': 16}}
```

`addSprints(sprint1,sprint2)` will return:

```
{'task1': {'John': 5, 'Rae': 10, 'Kelly': 18, 'Alex': 26, 'Mark': 5}, 'task2':  
{ 'Rae': 14, 'Alex': 4, 'Aaron': 25, 'Mark': 2}, 'task3': {'Kelly': 5, 'Alex': 1,  
'Ethan': 12, 'Helen': 10}, 'task4': {'Helen': 16}}
```

c) `addNLogs(logList)` – 10%

Now assume that the team recorded the log data for several sprints as a list of dictionaries. This list includes a dictionary for each development sprint.

Define a function `addNLogs` which takes a list of developer sprint logs and returns a dictionary which includes the collection of all project tasks and the total number of hours each developer has worked on each task. **Your function definition should use the Python `map` and `reduce` functions as well as the `sprintLog` and `addSprints` functions you defined in part(a) and (b).**

Example: Assume that the project had 3 sprints.

```
logList = [{'John': {'task1':5}, 'Rae': {'task1':10, 'task2':4}, 'Kelly':  
{ 'task1':8, 'task3':5}, 'Alex': {'task1':11, 'task2':2, 'task3':1},  
'Aaron': {'task2':15}, 'Ethan': {'task3':12}, 'Helen': {'task3':10}},  
{ 'Mark': {'task1':5, 'task2':2}, 'Kelly': {'task1':10}, 'Alex': {'task1':15,  
'task2':2}, 'Rae': {'task2':10}, 'Aaron': {'task2':10}, 'Helen': {'task4':16}},  
{ 'Alex': {'task3':10, 'task2':5, 'task4':6}, 'Rae': {'task3':5, 'task5':16},  
'Mark': {'task4':20}, 'Kelly': {'task2':5, 'task3':10, 'task4':12}, 'Helen':  
{ 'task5':10, 'task4':8}}]
```

For the above list `addNLogs` will return:

```
{'task1': {'John': 5, 'Rae': 10, 'Kelly': 18, 'Alex': 26, 'Mark': 5},  
'task2': {'Rae': 14, 'Alex': 9, 'Aaron': 25, 'Mark': 2, 'Kelly': 5},  
'task3': {'Kelly': 15, 'Alex': 11, 'Ethan': 12, 'Helen': 10, 'Rae': 5},  
'task4': {'Helen': 24, 'Alex': 6, 'Mark': 20, 'Kelly': 12},  
'task5': {'Rae': 16, 'Helen': 10}}
```

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

You can start with the following code:

```
def addNLogs (logList):  
    #write your code here
```

2. Dictionaries and lists

(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.)

3. (Lists) unzip – 12%

Write a function `unzip` that calculates the reverse of the zip operation. `unzip` takes a list of 3-tuples, `L`, as input and returns a tuple of lists, where each list includes the first, second, or third element from each tuple, respectively. Give a solution using higher order functions (`map`, `reduce` or `filter`), without using loops.

For example:

```

unzip ([ (1,"a",{1:"a"}), (2,"b",{2:"b"}), (3,"c",{3:"c"}), (4,"d",{4:"d"}) ])
returns
([1, 2, 3, 4], ['a', 'b', 'c', 'd'], [{1: 'a'}, {2: 'b'}, {3: 'c'}, {4: 'd'}])

```

You can start with the following code:

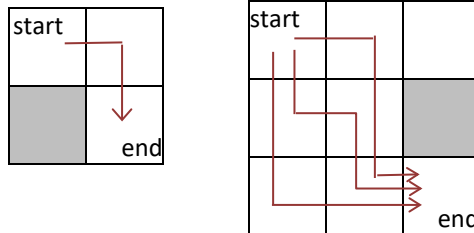
```

def unzip(L):
    #write your code here

```

4. (Recursion) numPaths(m,n,blocks) – 10%

Consider a robot in a $m \times n$ grid who is only capable of moving right or down in the grid (can't move left, up or diagonal). The robot starts at the top left corner, (1,1), and is supposed to reach to the bottom right corner: (m,n). Some of the cells in the grid are blocked and the robot is not allowed to visit those cells. Write a function numPaths that takes the grid length and width (i.e., m,n) and the list of the blocked cells (blocks) as argument and returns the number of different paths the robot can take from the start to the end. Give and answer using recursion. (A correct solution without recursion will be worth half the points.)



The blocked cells are represented as a list of (x,y) pairs where x is the row and y is column where the blocked cell is located.

For example, the 2x2 grid above has a blocked cell at (2,1). There is only one way for the robot to move from the start to the goal.

For the 3x3 grid above, the robot has 3 different paths.

```
numPaths(2,2,[(2,1)]) returns 1
numPaths(3,3,[(2,3)]) returns 3
numPaths(4,3,[(2,2)]) returns 4
numPaths(10,3,[(2,2),(7,1)]) returns 27
```

You can start with the following code:

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

5. Iterators

a) iterFile() – 25%

Create an iterator that represents the sequence of words read from a text file. The iterator is initialized with the name of the file and the iterator will return the next word from the file for each call to `__next__()`. The iterator should ignore all empty lines and end of line characters.

A sample input file ("testfile.txt") is attached.

For example:

```
mywords = iterFile("testfile.txt")
mywords.__next__() # returns CptS
mywords.__next__() # returns 355
mywords.__next__() # returns Assignment
for word in mywords:
    print(word)

# prints the rest of the words. See the Appendix for the output of the above
print statement.
```

You can start with the following code:

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

Important note: Your `iterFile` implementation should read the lines (or words) from the input text file as needed. An implementation that reads the complete file and dumps all words to a list all at once will be worth only 10 points.

b) `wordHistogram(words)` – 3%

Define a function `wordHistogram` that takes an iterator “words” (representing a sequence of words) and builds a histogram showing how many times each word appears in the file.

`wordHistogram` should return a list of tuples, where each tuple includes a unique word and the number of times that word appears in the file.

For example:

```
wordHistogram(iterFile("testfile.txt"))
```

returns

```
[('-', 5), ('CptS', 3), ('355', 3), ('Assignment', 3), ('3', 3), ('Python', 3),  
('Warmup', 3), ('text', 2), ('for', 2), ('This', 1), ('is', 1), ('a', 1),  
('test', 1), ('file', 1), ('With', 1), ('some', 1), ('repeated', 1), ('.', 1)]
```

You can start with the following code:

```
def wordHistogram(words):  
    #write your code here
```

Testing your functions

We will be using the `unittest` Python testing framework in this assignment. See <https://docs.python.org/2/library/unittest.html> for additional documentation.

The file `HW3SampleTests.py` provides some sample test cases comparing the actual output with the expected (correct) output for some problems. This file imports the `HW3` module (`HW3.py` file) which will include your implementations of the given problems.

You are expected to add **at least 2 more test cases** for each problem. Make sure that your test inputs cover all boundary cases. Choose test input different than those provided in the assignment prompt.

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

```
python -m unittest HW3SampleTests
```

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

```
python -m unittest -v HW3SampleTests
```

If you don’t add new test cases you will be deduced at least 5% in this homework.

Main Program

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...
```

Appendix

```
mywords = iterFile("testfile.txt")  
mywords.__next__() # returns CptS  
mywords.__next__() # returns 355  
mywords.__next__() # returns Assignment  
for word in mywords:  
    print(word)  
# prints the rest of the words
```

In problem 5, the above code will print the following:

```
3  
-  
Python  
Warmup  
This  
is  
a  
text  
test  
file  
for  
CptS  
355  
-  
Assignment  
3  
-  
Python  
Warmup  
With  
some  
repeated  
text  
for  
CptS  
355  
-  
Assignment  
3  
-  
Python  
Warmup  
.
```