CptS355 - Assignment 3 (Python)
Fall 2018

Python Warm-up

Assigned: Tuesday, October 9, 2018

Due: Thursday, October 18, 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 3 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 for each course you are enrolled in daily. You maintain the log of your hours in a Python dictionary as follows:

```
{'355':{'Mon':3,'Wed':2,'Sat':2},'360':{'Mon':3,'Tue':2,'Wed':2,'Fri':1
0},'321':{'Tue':2,'Wed':2,'Thu':3},'322':{'Tue':1,'Thu':5,'Sat':2}}
```

The keys of the dictionary are the course numbers and the values are the dictionaries which include the number of hours you studies on a particular day of the week. Please note that you may not study for some courses on some days OR you may not study for a particular course at all.

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

```
{'Fri': 10, 'Mon': 6, 'Sat': 4, 'Thu': 8, 'Tue': 5, 'Wed': 6}
```

(*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 your courses 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 course log dictionaries and returns a dictionary which includes the total number of hours that you have studied on each day of the week 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.

```
[{'355':{'Mon':3,'Wed':2,'Sat':2},'360':{'Mon':3,'Tue':2,'Wed':2,'Fri':
10},'321':{'Tue':2,'Wed':2,'Thu':3},'322':{'Tue':1,'Thu':5,'Sat':2}},
{'322':{'Mon':2},'360':{'Thu':2, 'Fri':5},'321':{'Mon':1, 'Sat':3}},
{'355':{'Sun':8},'360':{'Fri':5},'321':{'Mon':4},'322':{'Sat':3}}]
```

For the above dictionary `addDictN` will return:

```
{'Fri': 20,'Mon': 13,'Sat': 10,'Sun': 8, 'Thu': 10, 'Tue': 5, 'Wed': 6}
```

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

2. 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:

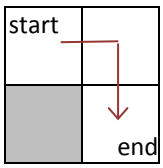
```
L2 = [(0,{"x":0,"y":True,"z":"zero"}),
      (0,{"x":1}),
      (1,{"y":False}),
      (1,{"x":3, "z":"three"}),
      (2,{})]

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

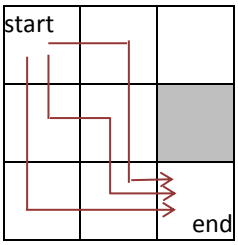
(*Note*: I suggest you to provide a recursive solution to this problem.
Hint: Define a helper function with an additional parameter that hold the list index which will be searched in the next recursive call.)

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

Consider a robot in a `mXn` 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 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, 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

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

4. **Fun stuff - 15%**

Write a function, `palindromes`, which takes a string as input and returns a list of the unique palindromes that appear in the input string. You may assume that the input string doesn't have any 'space' characters and all characters are lowercase. The strings in the output should be sorted alphabetically.
(Palindrome is a string that reads the same backward as forward, e.g., madam or kayak).

`palindromes ('cabbbaccab')` returns
`['abbba', 'acca', 'baccab', 'bb', 'bbb', 'cabbbac', 'cc']`

`palindromes ('bacdcabdbacdc')` returns
`['abdba', 'acdca', 'bacdcab', 'bdb', 'cabdbac', 'cdc', 'cdcabdbacdc', 'dcabdbacd']`

`palindromes ('myracecars')` returns
`['aceca', 'cec', 'racecar']`

You can start with the following code:

```
def palindromes(S):
      #write your code here

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

5. Iterators

a) `iterApply()` – 10%

Create an iterator whose constructor takes an integer (`n`) and a function (`f`) as argument and represents the sequence of the numbers `f(n),f(n+1),f(n+2),….`

For example:

```
>>> squares = iterApply(1,lambda x: x**2)
>>> squares.__next__()
1
>>> squares.__next__()
4
>>> squares.__next__()
9
```

You can start with the following code:

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

b) `iMerge(iNumbers1,iNumbers2,N)` – 10%

Define a function `iMerge` that takes 2 iterable values "`iNumbers1`" and "`iNumbers2`" (which are sorted sequences of increasing numbers), and merges the two input sequences. `iMerge` returns the first `N` elements from the merged sequence. The numbers in the merged sequence needs to be sorted as well. (Note that the iterator retrieves the next element in the sequence with the second call to `iMerge`. You may define a `__prev__` method in `iterApply` if you need to reposition the iterator to the previous element in the sequence.)

For example:

```
>>> squares = iterApply(1,lambda x: x**2)
>>> triples = iterApply(1,lambda x: x**3)

>>> iMerge(squares,triples,8)
[1, 1, 4, 8, 9, 16, 25, 27]
>>> iMerge(squares,triples,10)
[36, 49, 64, 64, 81, 100, 121, 125, 144, 169]
>>> iMerge(squares,triples,6)
[196, 216, 225, 256, 289, 324]
```

You can start with the following code:

```
def iMerge(iNumbers1,iNumbers2,N):
    #write your code here

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

6. Streams

a) `streamRandoms(k,min,max)` – 10%

Using the `Stream` class we defined in the lecture, write a function `streamRandoms(k,min,max)` that creates an infinite stream of positive random integers starting at `k` (i.e., make the first element of the stream `k`). The values in the stream should be randomly generated and each value should be between `min` and `max` argument values (inclusive).

(You may make use of the *`random.randint(a,b)`* method to generate random integers. To get access to the random module, you need to add the line "`import random`" in your program.)

For example:

```
>>> rStream = streamRandoms(1,1,100)
>>> myList = []
>>> for i in range (0,10):
        myList.append(rStream.first)
        rStream = rStream.rest
```

After running the above code, `myList` should include the first 10 values of the `rStream`. (for example: `[1, 28, 2, 33, 72, 96, 1, 85, 56, 31]`)

You can start with the following code:

```
def streamRandoms (k,min,max):
    #write your code here
```

b) oddStream(stream) – 10%

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

For example:

```
>>> oddS = oddStream(streamRandoms(1,1,100))
>>> myList = []
>>> for i in range(0,100):
        myList.append(oddS.first)
        oddS =oddS.rest
```

`myList` should include 100 odd integers ranging between 1 and 100.

You can start with the following code:

```
def oddStream(stream):
    #write your code here

def testoddStream ():
    #write your code here;
```

Test your code:

Here is an example test function for testing `addDict`. Make sure to include 2 test cases for each function. You don't need to write test functions for questions 5 and 6.

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

def testaddDict():
    d = {'355':{'Mon':3,'Wed':2,'Sat':2},'360':{'Mon':3,'Tue':2,'Wed':2,'Fri':10},
         '321':{'Tue':2,'Wed':2,'Thu':3},'322':{'Tue':1,'Thu':5,'Sat':2}}
    if addDict({}) != {}:
        return False
    if dict(sorted(list(addDict(d).items()))) != {'Fri': 10, 'Mon': 6, 'Sat': 4,
         'Thu': 8, 'Tue': 5, 'Wed': 6}:
        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 testaddDict():
       print ( passedMsg % 'addDict' )
    else:
       print ( failedMsg % 'addDict')
    # etc. for the other tests.
    # notice how you are repeating a lot of code here
    # think about how you could avoid that
```