CptS355 - Assignment 3 - Fall 2021

Python Warm-up

Assigned: Thursday, October21, 2021

Weight: This assignment will count for 6.5% 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`**. 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. In addition, rename the **`HW3SampleTests.py`** file as **`HW3Tests.py`** and add your own test cases in this file. You are expected to add one more test case for each problem. Choose test inputs different than those provided in the assignment prompt. When you are done and certain that everything is working correctly, turn in your files by uploading on the Assignment-3(Python) DROPBOX on Canvas. The files that you upload must be named **`HW3.py`** and **`HW3Tests.py`**. Please don't zip your code; directly attach the .py files to the dropbox. You may turn in your assignment up to 3 times. Only the last one submitted will be graded. Implement your code for Python3.

Grading

The assignment will be marked for good programming style as well as thoroughness of testing and clean and correct execution. **6% of the points will be reserved for the test functions.** 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",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. merge_by_year, merge_year, and getmax_of_flavor

Consider the cat feeding log data we used in Haskell class exercises. We re-format the same data as a Python dictionary where keys are the timestamps (month, year pairs) and values are the feeding logs.

```
my_cats_log = {
  {(2,2019):{"Oceanfish":7, "Tuna":1, "Whitefish":3, "Chicken":4, "Beef":2},
  (5,2019):{"Oceanfish":6, "Tuna":2, "Whitefish":1, "Salmon":3, "Chicken":6},
  (9,2019):{"Tuna":3, "Whitefish":3, "Salmon":2, "Chicken":5, "Beef":2, "Turkey":1, "Sardines":1},
  (5,2020):{"Whitefish":5, "Sardines":3, "Chicken":7, "Beef":3},
  (8,2020):{"Oceanfish":3, "Tuna":2, "Whitefish":2, "Salmon":2, "Chicken":4, "Beef":2, "Turkey":1},
  (10,2020):{"Tuna":2, "Whitefish":2, "Salmon":2, "Chicken":4, "Beef":2, "Turkey":4, "Sardines":1},
  (12,2020):{"Chicken":7,"Beef":3, "Turkey":4, "Whitefish":1, "Sardines":2},
  (4,2021):{"Salmon":2,"Whitefish":4, "Turkey":2, "Beef":4, "Tuna":3, "MixedGrill": 2},
  (5,2021):{"Tuna":5,"Beef":4, "Scallop":4, "Chicken":3},
  (6,2021):{"Turkey":2,"Salmon":2, "Scallop":5, "Oceanfish":5, "Sardines":3},
  (9,2021):{"Chicken":8,"Beef":6},
  (10,2021):{ "Sardines":1, "Tuna":2, "Whitefish":2, "Salmon":2, "Chicken":4, "Beef":2, "Turkey":4}
}
```

Assume, you would like to create a summary of the data and sum the number cans for each flavor within each year. For example, when you aggregate the above dictionary, you will get the following:

```
{
2019:{'Oceanfish':13,'Tuna':6,'Whitefish':7,'Chicken':15,'Beef':4,'Salmon':5,'Turkey':1,'Sardines':1},
2020:{'Whitefish':10,'Sardines':6,'Chicken':22,'Beef':10,'Oceanfish':3,'Tuna':4,'Salmon':4,'Turkey':9},
2021:{'Salmon':6,'Whitefish':6,'Turkey':8,'Beef':16,'Tuna':10,'MixedGrill':2,'Scallop':9,'Chicken':15,
      'Oceanfish':5, 'Sardines':4}
}
```

a) merge_by_year(feeding_log) – 10%

Define a function `merge_by_year` that aggregates the feeding log data as described above. Your function should not hardcode the cat food flavors and timestamps. You may use loops in your solution.
The items in the output dictionary can have arbitrary order.

`merge_by_year(my_cats_log)` returns the above dictionary.

You can start with the following code:

```
def merge_by_year(feeding_log):
    #write your code here
```

Important Notes:

1. Your function should not change the input dictionary value.
2. You should not hardcode the dictionary keys in your code.

b) merge_year(feeding_log, year) – 15%

Now consider that you would like to sum up the number of cans for each flavor for a given year – i.e., the "year" argument. For example:

```
merge_year(my_cats_log, 2019) returns
{'Oceanfish':13,'Tuna':6,'Whitefish':7,'Chicken':15,'Beef':4,'Salmon':5,
'Turkey':1,'Sardines':1}
merge_year(my_cats_log, 2021) returns
{'Salmon':6,'Whitefish':6,'Turkey':8,'Beef':16,'Tuna':10,'MixedGrill': 2,
'Scallop':9,'Chicken':15,'Oceanfish':5,'Sardines':4}
```

Define the function `merge_year` that aggregates the feeding log data for the given year. **Your function definition should not use loops or recursion but use the Python `map`, `reduce`, and/or `filter` functions**. You may define and call helper (or anonymous) functions, however your helper functions should not use loops or recursion. You will not get any points if your solution (or helper functions) uses a loop. If you are using reduce, make sure to import it from `functools`.

You can start with the following code:

```
def merge_year(feeding_log, year):
    #write your code here
```

Important Notes:

1. You are not allowed to use list or dictionary comprehension solution that involves a loop.
2. You are not allowed to use Python Combinators or other Python libraries we haven't covered in class.
3. The instructor will provide an example function (in class) which combines 2 dictionaries using map, reduce and filter. You may use that function in your solution.

c) getmax_of_flavor(feeding_log, flavor) – 15%

Assume you would like to find the maximum number of cans of some specific flavor you fed to your cat in a single month. Define a function "getmax_of_flavor" that takes the feeding log and a flavor name as argument, and it returns a tuple that includes the month having max number of cans and the number of cans value. For example:

```
getmax_of_flavor(my_cats_log, "Tuna") returns ((5, 2021), 5) #i.e., you fed the
   max number of "Tuna" cans to your cat in May 2021, which is 5 cans.

getmax_of_flavor(my_cats_log, "Beef") returns ((9, 2021), 6)
```

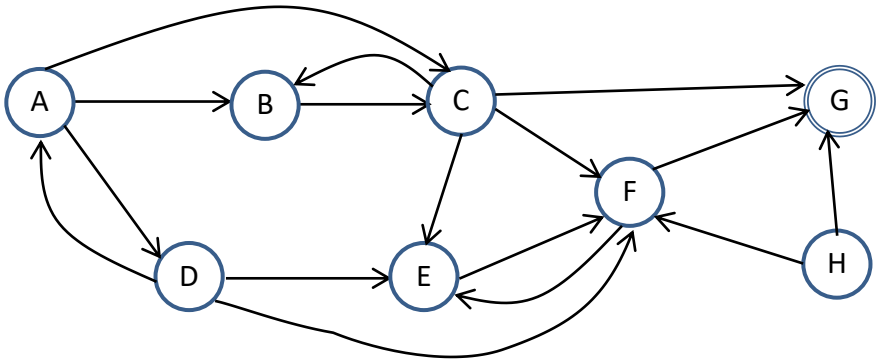
Your function definition should not use loops or recursion but use the Python `map`, `reduce`, and/or `filter` functions. You may define and call helper (or anonymous) functions, however your helper functions should not use loops or recursion. You will not get any points if your solution (or helper functions) uses a loop. If you are using reduce, make sure to import it from `functools`.

You can start with the following code:

```
def getmax_of_flavor(feeding_log, flavor):
    #write your code here
```

2. follow_the_follower(graph) – 16%

Consider the following directed graph where each node has zero or more outgoing edges. Assume the graph nodes are assigned unique labels. This graph can be represented as a Python dictionary where the keys are the starting nodes of the edges and the values are the set of the ending nodes (represented as Python sets). Note that some nodes in the graph are halting nodes, i.e., they don't have any outgoing edges. Those nodes are marked with double lines in the graph.




```
{'A':{'B','C','D'},'B':{'C'},'C':{'B','E','F','G'},'D':{'A','E','F'},'E':{'F'},
'F':{'E', 'G'},'G':{}, 'H':{'F','G'}}
```

a) follow_the_follower(graph) – 10%

Write a function, `follow_the_follower`, which takes a graph dictionary as input and returns the pair of nodes that are connected with each other through direct edges in both directions. For example, the pair ('A','D') is connected in both directions; there is an edge from 'A' to 'D' and an edge from 'D' to 'A'.
The output is a list of tuples where each tuple include the labels of such pairs.
For example:

```
graph = {'A':{'B','C','D'}, 'B':{'C'}, 'C':{'B','E','F','G'}, 'D':{'A','E','F'},
'E':{'F'}, 'F':{'E', 'G'}, 'G':{}, 'H':{'F','G'}}
```

```
follow_the_follower(graph)
returns  [('A','D'), ('B','C'), ('C','B'), ('D','A'), ('E','F'), ('F','E')]
```

You don't need to remove the swapped duplicates in the output. The tuples in the output can be in arbitrary order.

You can start with the following code:

```
def follow_the_follower(graph):
    #write your code here
```

b) follow_the_follower2(graph) – 6%

Re-write your `follow_the_follower` function using list comprehension. Name this function `follow_the_follower2`.

You can start with the following code:

```
def follow_the_follower2(graph):
    #write your code here
```

3. connected(graph,node1,node2) – 15%

Consider the graph dictionary in question 2. Write a recursive function, `connected`, which takes a graph dictionary and two node labels as input and returns `True` if the given nodes are connected in the graph through some path. It returns `False` otherwise. For example, `connected(graph,'A','F')` will return `True`, since they are connected through the path `'A','B','C','F'`. Also, `connected(graph,'E','A')` will return `False`, since there is no path between those nodes.

```
graph = {'A':{'B','C','D'}, 'B':{'C'}, 'C':{'B','E','F','G'}, 'D':{'A','E','F'},
'E':{'F'}, 'F':{'E', 'G'}, 'G':{}, 'H':{'F','G'}}
```

```
debug(connected(graph, 'A' ,'F') ) returns True
debug(connected(graph, 'E' ,'A') ) returns False
debug(connected(graph, 'A' ,'H') ) returns False
debug(connected(graph, 'H' ,'E') ) returns True
```

You can start with the following code:

```
def connected(graph, node1, node2):
    #write your code here
```

4. Iterators

a) lazy_word_reader() – 20%

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, i.e., '\n' .
A sample input file (`"testfile.txt"`) is attached to the Canvas dropbox.

For example:

```
mywords = lazy_word_reader("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
```

You can start with the following code:

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

Important Note: Your `lazy_word_reader` 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 5 points.

b) word_histogram(words) – 3%

Define a function word_histogram that takes a "lazy_word_reader" iterator object as input and builds a histogram showing how many times each word appears in the sequence (i.e., the file that the iterator reads from). word_histogram should return a list of tuples, where each tuple includes a unique word and the number of times that word appears in the file.

The tuples in the output should be sorted based on the descending order of the word counts. The tuples that have the same word count should be further sorted based on the words (in ascending order).

For example:

```
word_histogram (lazy_word_reader ("testfile.txt"))
```

returns

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

You can start with the following code:

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

Testing your functions (6%)

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 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.

Rename the HW3SampleTests.py file as HW3Tests.py and add your own test cases in this file. You are expected to add **at least one more test case** 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 6% in this homework.

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 = lazy_word_reader("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
```

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
.
dot
```