CptS355 - Assignment 3 - Fall 2023

Python Warm-up

Assigned: Friday, October 20, 2023

Weight: This assignment will count for 3.5% 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. **Around 5% 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.

Problems:

1. (Dictionaries)

a) `cat_diet(feeding_log)` – 10%

Assume your cat gained too much weight and your veterinarian recommended you keep track of the number of cat food cans you feed to your pet every month. In addition, they recommended you balance their diet and avoid giving the same type of food. So, you start recording how many cans you feed your cat in a Python dictionary. This dictionary logs the number of cans your cat consumed from each flavor each month.

```
myCatsLog = {
    'Beef': {"Jul": 2, "Sep": 2, "Oct": 3, "Nov": 2, "Dec": 2, "June": 3},
    'Chicken': {"Jul": 4, "Aug": 6, "Sep": 5, "Oct": 7, "Nov": 4, "Dec": 4, "June": 7},
    'Oceanfish': {"Jul": 7, "Aug": 6, "Nov": 3},
    'Salmon': {"Aug": 3, "Sep": 2, "Nov": 2, "Dec": 2},
    'Sardines': {"Sep": 1, "Oct": 3, "Dec": 1, "June": 2},
    'Tuna': {"Jul": 1, "Aug": 2, "Sep": 3, "Nov": 2, "Dec": 2},
    'Turkey': {"Sep": 1, "Nov": 1, "Dec": 4, "June": 4},
    'Whitefish': {"Jul": 3, "Aug": 1, "Sep": 3, "Oct": 5, "Nov": 2, "Dec": 2, "June": 1}}
```

Assume, we would like to re-organize the data and create a dictionary that includes months as keys and the dictionary of flavors and their counts as values. For example, when you organize the above dictionary, you will get the following:

```
{ "June":{"Chicken":7,"Beef":3, "Turkey":4, "Whitefish":1, "Sardines":2},
  "Jul":{"Oceanfish":7, "Tuna":1, "Whitefish":3, "Chicken":4, "Beef":2},
  "Aug":{"Oceanfish":6, "Tuna":2, "Whitefish":1, "Salmon":3, "Chicken":6},
  "Sep":{"Tuna":3, "Whitefish":3, "Salmon":2, "Chicken":5, "Beef":2, "Turkey":1, "Sardines":1},
  "Oct":{"Whitefish":5, "Sardines":3, "Chicken":7, "Beef":3},
  "Nov":{"Oceanfish":3, "Tuna":2, "Whitefish":2, "Salmon":2, "Chicken":4, "Beef":2, "Turkey":1},
  "Dec":{"Tuna":2, "Whitefish":2, "Salmon":2, "Chicken":4, "Beef":2, "Turkey":4, "Sardines":1}
}
```

Define a function `cat_diet` that organizes the feeding log data as described above. Your function should not hardcode the cat food flavors and months.
(The items in the output dictionary can have arbitrary order.)

b) `cats_favorite(feeding_log)` – 15%

Assume, you would like to find your cat's most popular canned food flavor, i.e., the flavor that has the max total number of cans. Define a function "`cats_favorite`" that will return the popular flavor and its total can count. For example:

```
cats_favorite (myCatsLog) returns ('Chicken', 37)
#i.e., chicken has the max number of total cans among all food flavors,
 which is 37.
```

Your function definition should not use loops or recursion but use the Python `map` and `reduce` functions. You may define and call helper (or anonymous) functions, however your helper functions should not use loops or recursion. If you are using reduce, make sure to import it from `functools`.

2. (Lists) unzip(L) – 15%

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

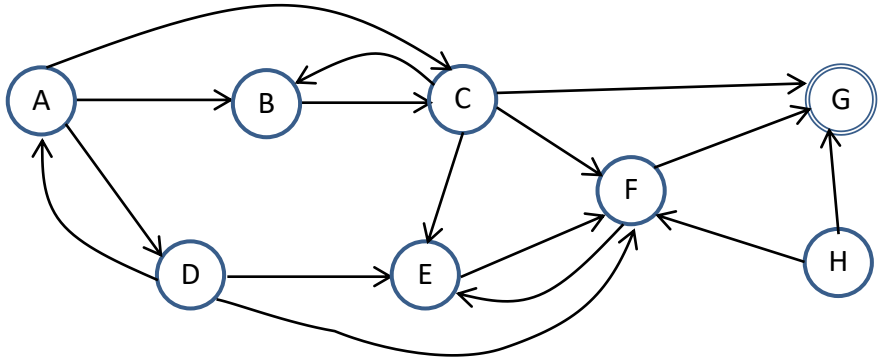
```
unzip ([(1,"a",10),(2,"b",11),(3,"c",12),(4,"d",13)])
```

returns

```
([1, 2, 3, 4], ['a', 'b', 'c', 'd'], [10, 11, 12, 13])
```

3. (Dictionaries)

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) connected(graph) – 15%

Write a function, `connected`, 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'}}

connected(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.

b) connected2(graph) – 10%

Re-write your `connected` function using list comprehension. Name this function `connected2`.

(c) has_path(graph,node1,node2) – 15%

Consider the graph dictionary in question 2. Write a recursive function, `has_path`, 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, `has_path(graph,'A','F')` will return `True`, since they are connected through the path `'A','B','C','F'`. Also, `has_path(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'}}
```

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

4. (Iterator) str_to_list – 15%

Write a function `str_to_list` that takes a Python string value as input and converts it to a nested list where the characters between matching parentheses are included in sublists. The output list should have the same nested structure of the parentheses of the input string. You can assume that the parentheses in the input string are correct and there are no unmatching opening or closing parenthesis.

Hints: Give a recursive solution. Convert the input string to an iterator using the `iter()` function and pass it to your recursive function. This function should recursively parse the nested parentheses in the string. We will talk more about this solution approach in class.

For example:

```
str_to_list( "ab(c(def)(gh(ijk(lm)nop)qr)st(uv)x" )
```

returns

```
['a','b',['c',['d','e','f'],['g','h',['i','j','k',['l','m'],'n','o','p'],'q','r'],'s','
t',['u','v'],'x']]
```

```
str_to_list( "1()(((2))())()(3)4" )
```

returns

```
['1', [], [[['2']], []], [], ['3'], '4']
```

Testing your functions (5%)

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. The test methods for your new tests are already included in the `HW3SampleTests.py` file; you just need to add your test inputs and the appropriate assertion conditions to these methods.

You are expected to add **at least one more test case** for each problem (except problems 1(a) and 1(b)). 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.

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...
```