Homework 6 - Dictionaries and Try/Except

CS 1301 - Intro to Computing - Spring 2020

Important

- Due Date: **Tuesday, February 25th, 11:59 PM.**
- This is an individual assignment. High-level collaboration is encouraged, **but your submission must be uniquely yours.**
- Resources:
  - TA Helpdesk
  - Email TA's or use class Piazza
  - How to Think Like a Computer Scientist
  - CS 1301 YouTube Channel
- Comment out or delete all function calls. Only import statements, global variables, and comments are okay to be outside of your functions.
- **Read the entire document before starting this assignment.**

Responsible Coding: Some functions in this assignment will have a 'Responsible Coding' badge next to them. These functions are written to help you think about how you can use your skills as a programmer to analyze problems related to *sustainability*, such as ethics, health, and the environment.

Hidden Test Cases: In an effort to encourage debugging and writing robust code, we will be including hidden test cases on Gradescope for some functions. You will not be able to see the input or output to these cases. Below is what you will see in Gradescope, please disregard it:

```
Test failed: False is not true
```

Try/Except: Try/Except blocks are very useful if the code you are trying to execute can potentially throw errors. These blocks allow you to handle these errors **without** stopping the execution of the program. You will see that this nicely complements dictionaries as we might try to access a key that does not exist in a dictionary. If that happens, we may want to do something else instead of terminate our program.

Written by Damian Henry (dhenry15@gatech.edu) & Rajit Khanna (rkhanna23@gatech.edu)

Average Scores

Function Name: average_scores
Parameters: dictionary (`dict`) of exams (`str`) mapped to scores (`list`)
Returns: list of tuples where each exam is paired with its average score (`list`)
Description: You are given a dictionary where the keys are exams and the values are the scores for each exam:

```
{
    exam1: [score1, score2, score3, etc.],
    exam2: [score1, score2, etc.],
    exam3: [score1, score2, score3, score4, etc.]
    .
    .
    .
}
```

Return a list of tuples where each tuple has the exam name and the average score for that exam rounded to 2 decimal places:

```
[(exam1, avg1), (exam2, avg2), (exam3, avg3), etc.]
```

Note: If the exam has not been taken yet, the average is None. Try/except might be useful here.

Test Cases

```
>>> exam_info = {"Test 1" : [70.5, 95.5, 85.5], "Test 2": [88.0, 97.5, 76.0], "
Test 3": [100.0]}
>>> print (average_scores(exam_info))
[("Test 1", 83.83), ("Test 2", 87.17), ("Test 3", 100.0)]
```

```
>>> exam_info = {"Exam 1": [70.0, 80.0, 90.0], "Exam 2" : [85.0, 85.0, 90.0,  9
5.0], "Exam 3" : [] }
>>> print (average_scores(exam_info))
[("Exam 1", 80.0), ("Exam 2", 88.75), ("Exam 3", None)]
```

Class Finder

Function Name: class_finder
Parameters: courseDict (`dict`), friendsList (`list`)
Returns: goodClasses (`list`)
Description: Write a function that takes in a list of your friends and a dictionary that maps the

names of different courses as strings to lists of the course rosters. The function should return a list of the courses that have at least two of your friends in them.

The format of courseDict will be as follows:

```
{
    course1: [person1, person2, . . . ],

    course2: [person1, person2, . . . ],

    . . .

}
```

Hint: The python `in` operator might be useful!

```
>>> courseDict = {"CS1301": ["Damian", "Peter", "Jasmine C.", "Jakob", "Cathy"]
, "CS2110" : ["Arushi", "Brae", "Arvin", "Cathy", "Jasmine Y.", "Caitlin"], "CS
1332": ["Damien", "Jakob", "Juliette", "Jasmine Y." "Jasmine C.", "Arvin"]}
>>> friendsList = ["Damian", "Peter", "Brae", "Arushi"]
>>> print(class_finder(courseDict, friendsList))

['CS1301', 'CS2110']
```

```
>>> courseDict = {"MATH2550": ["Jakob", "Peter", "Arushi", "Anthony", "Arvin"],
 "PSYC1101": ["Anthony", "Peter", "Arvin", "Juliette", "Brae", "Jasmine Y."], "
EAS1600": ["Anthony", "Jasmine C.", "Jakob", "Damien", "Caitlin"], "APPH1040":
["Rajit", "Peter", "Jasmine Y.", "Cathy", "Damien", "Arushi"]}
>>> friendsList = ["Jakob", "Jasmine Y.", "Anthony"]
>>> print(class_finder(courseDict, friendsList))
['MATH2550', 'PSYC1101', 'EAS1600']
```

Security Clearance Responsible Coding

Function Name: security_clearance
Parameters: a list of tuples (`list`)
Returns: dictionary of organized security clearance information (`dict`)

Description: As computer scientists, we should always keep security in mind. We don't want unauthorized users accessing restricted files. The higher-ups give you a list of security information for each personnel:

`[(kind,level,name), (kind,level,name),  (kind,level,name) etc]`

Your mission is to organize this list so that all of those with the same kind and level are grouped together for easier look up. The higher-ups specify that the format must be:

```
{kind1: {
            level1: [name2, name3, name6, etc.],
            level2: [name1, name4, etc.],
                .
                .
                .
        },
 kind2: {
            level2: [name7, name8, etc.]
                .
                .
                .
        }
     .
     .
     .
}
```

Test Cases

```
>>> info = [("Party", 1, "Literally everyone"), ("Party", 2, "Who do you know h
ere?"), ("Party", 3, "Brothers only")]
>>> print (security_clearance(info))
{"Party" : {1 : ["Literally everyone"], 2: ["Who do you know here?"], 3: "Broth
ers only"}}
```

```
>>> info = [("Top Secret", 3, "Drake"), ("Confidential", 2, "Shawn Mendes"), ("
Unclassified", 1, "Pusha T"), ("Top Secret", 3, "LeBron James"), ("Confidential
", 1, "Future"), ("Confidential", 1, "Arizona Zervas")]
>>> print (security_clearance(info))
{"Top Secret" : {3: ["Drake", "LeBron James"]}, "Confidential" : {2: ["Shawn Me
ndes"], 1 : ["Future", "Arizona Zervas"]}, "Unclassified" : {1 : ["Pusha T"]}}
```

Find Me

Function Name: find_me
Parameters: a dictionary of strings mapping to strings (`dict`), person1 (`str`), person2 (`str`)
Returns: number of people in between the first and last person (`int`)

Description: Given a dictionary and two names, find the number of people in between them. The dictionary maps a person (`key`) to the person (`value`) in front of them in a line. If `person2` is the value of the `person1` key, then they have 0 people in between them. If there is no path between the two people, return None.

Hint: A while loop to check if the current person you are on is actually in the dictionary might be helpful. However, this also can be done with a for loop. You pick!

Test Cases

```
>>> people = {"Damian": "Jakob", "Rajit": "Jasmine Y.", "Jasmine Y.": "Arvin",
"Caitlin": "Cathy", "Jakob": "Caitlin"}
>>> person1 = "Damian"
>>> person2 = "Cathy"
>>> print (find_me(people, person1, person2))
2
# "Damian" -> "Jakob" -> "Caitlin" -> "Cathy"
```

```
>>> people = {"Arushi": "Juliette", "Brae": "Arushi", "Anthony" : "Jasmine C."
}
>>> person1 = "Brae"
>>> person2 = "Jasmine C."
>>> print (find_me(people, person1, person2))
None
# "Brae" -> "Arushi" -> "Juliette" -/> oh no, the path is broken!
```

Error Finder

Function Name: error_finder
Parameters: numList (`list`), indexList (`list`)
Returns: (average, errorDict) (`tuple`)
Description: Write a function that takes in a list of numbers and a list of indices. Note that indexList may not only contain valid indices. The function should keep track of the number and type of errors that occur. Specifically, it should account for `IndexError` and `TypeError`. It should return the average of all the numbers at valid indices and a dictionary containing the number and type of errors together in a tuple. errorDict should be formatted as follows:

```
{"IndexError": 0, "TypeError": 0}
```

If none of the indices in the indexList are valid, the function should return 0 as the average.

```
>>> numList = [4,5,1,7,2,3,6]
>>> indexList = [0, "4", (1,), 18, "", 3, 5.0, 7.0, {}, 20]
>>> print(error_finder(numList, indexList))

(5.5, {'IndexError': 2, 'TypeError': 6})
```

```
>>> numList = [1, 0, 18, 22, 3, -1, 5, 4, 9]
>>> indexList = [(7,), 4.0, {True: False}, 0, [11], "not an index", 12, 2.5]
>>> print(error_finder(numList, indexList))

(1.0, {'IndexError': 1, 'TypeError': 6})
```

Grading Rubric

Function	Points
average_scores	20
class_finder	20
security_clearance	20
find_me	20
error_finder	20
Total	**100**

Provided

The `HW06.py` skeleton file has been provided to you. This is the file you will edit and implement. All instructions for what the functions should do are in this skeleton and this document.

Submission Process

For this homework, we will be using Gradescope for submissions and automatic grading. When you submit your `HW06.py` file to the appropriate assignment on Gradescope, the autograder will run automatically. The grade you see on Gradescope will be the grade you get, unless your grading TA sees signs of you trying to defeat the system in your code. You can resubmit this assignment an unlimited number of times until the deadline; just click the "Resubmit" button at the lower right-hand corner of Gradescope. You do not need to submit your `HW06.py` on Canvas.