Quiz 1

Note: Focus only on the **your_solution.py** file. Run the program with python3 **your_solution.py,** when the driver prompt appears press Enter and it will give you the results.
If not, then test by print by statements, it is your choice. Just comment out the driver section, if you would like to test on your own.
The Part1 is not tested by the driver, you will have to test it yourself.

Part 2 Notes:
You can pass a sort data structure into the sorted function, for example:
db = {'a': 1, 'b':2}
sorted(sorted(db), key = lambda t: t[1])
This fill first sort the db by default, then sort the sorted result using the key.

Instead of the **dict** you can also use **defaultdict** which we have not covered yet, but you can refer to the Python Documentation to see how it can be used.
Part 1

1. Define a function ***students_ gpas*** that returns a dictionary. Assume we have a list of students (***students***) and a list of their gpas (***gpas***). Create a dictionary whose first key is the first student in ***students*** and whose associated value is the first gpa in the ***gpas***; whose second key is the second student in ***students*** and whose associated value is the second gpa in the ***gpas*** ; etc. So if students = ['bob', 'carol', 'ted', 'alice'] and gpas = [3.0, 3.2, 2,8, 3.6] the resulting dict might be {'carol': 3.2, 'ted': 2, 'alice': 8, 'bob': 3.0}.
   *** Hint: use the dict constructor and the zip function.

2. Define a function runners_places that returns a dictionary. Assume that we have a list of runners in the order they finished the race. For example ['bob','carol','ted','alice']. Create a dictionary whose keys are the runners and whose values are the place in which they finished. For this example the resulting dict might be {'carol': 2, 'ted': 3, 'alice': 4, 'bob': 1}.
   *** Hint: use a comprehension and enumerate.

3. Assume that we have a list of values l and a function f, and we want to define a function that computes the value x in l whose f(x) is the smallest.
   For example, if l = [-2, -1, 0, 1, 2] and def f(x) : return abs(x), then calling ***min_ value(l,f)*** would return 0.
   *** Hint: try to write the min_v function in one line: a returns statement calling min, on a special comprehension, and indexing.

Part 2

A stock trading company stores a database that is a dictionary (dict) whose keys are client names (str); associated with each client name is a list of 3-tuples representing the transaction history of that client: in the 3-tuple,

the **first** index is a stock name (str),

the **second** index is the number of shares traded for that transaction (int: positive means buying shares; negative means selling shares), and

the **third** index is the price per share in dollars (int for simplicity) for that transaction.

The list shows the order in which the transactions occurred. A simple/small database can look like

{
'Carl': [('Intel', 30, 40), ('Dell', 20, 50), ('Intel', -10, 60), ('Apple', 20, 55)],
'Barb': [('Intel', 20, 40), ('Intel', -10, 45), ('IBM', 40, 30), ('Intel', -10, 35)],
'Alan': [('Intel', 20, 10), ('Dell', 10, 50), ('Apple', 80, 80), ('Dell', -10, 55)],
'Dawn': [('Apple', 40, 80), ('Apple', 40, 85), ('Apple', -40, 90)]
}

Try to define your functions only by using one return statement.

Hint: write these functions using multiple statements first, then if you can, transform that code into a single statement using combinations of **comprehensions** (sometimes with multiple loops) and calls to sort. No function should alter its argument.

• Use sequence unpacking, you can use _ for unneeded names

• In the notes, see how multiple statements can be rewritten as an equivalent statement comprehension.

•Sometimes use two loops in one comprehension. Complicated problems might require multiple comprehensions each with its own loop.

• For sorting problems, sometimes build the thing that must be sorted using a comprehension and call sorted on it, figuring out a key function; sometimes, sort something and then use the result in a comprehension to build what needs to be returned.

1. The ***stocks*** function takes a **database** (dict) as an argument and returns a set: all stock names traded. E.g., if db is the **database** above, calling stocks(db) returns {'Dell', 'Apple', 'Intel', 'IBM'}.
2. The ***clients_by_volume*** function takes a **database** (dict) as an argument and returns a list: client names (str) in decreasing order of their **volume** of trades (the sum of the number of shares they traded); if two clients have the same volume, they must appear in increasing alphabetical order. Important: selling 15 shares (appearing as -15 shares) counts as 15 shares traded: add up the absolute values of the number of shares traded. E.g., if db is the **database** above, calling clients_by_volume(db) returns ['Alan', 'Dawn', 'Barb', 'Carl']. Alan and Dawn each have a volume of 120 shares (they appear first in increasing alphabetical order); Barb and Carl each have a volume of 80 shares (they appear next in increasing alphabetical order).
3. The ***stocks_by_volume*** function takes a **database** (dict) as an argument and returns a list: 2- tuples of stock names (str) followed its **volume** (shares of that stock name traded among all clients) in decreasing order of volume; if two stocks have the same volume, they must appear in increasing alphabetical order. For example, if db is the **database** above, calling stocks_by_volume(db) returns [('Apple',220), ('Intel',100), ('Dell',40), ('IBM',40)]. Dell and IBM each had a volume of 40 shares (they appear last in increasing alphabetical order). Hint: you can use/call the stocks function you wrote in part 3a here.
4. Define the ***by_stock*** function, which takes a **database** (dict) as an argument; it returns a dictionary (dict) associating a **stock** name with an inner dictionary (dict) of all the **clients** (str) as keys associated with their **trades** for that **stock** (a list of 2-tuples of int). E.g., if db is the **database** above, calling by_stock(db) returns a dictionary whose contents are

```
{
'Intel': {'Carl': [(30, 40), (-10, 60)], 'Barb': [(20, 40), (-10, 45), (-10, 35)],  'Alan': [(20, 10)]},
'Dell': {'Carl': [(20, 50)], 'Alan': [(10, 50), (-10, 55)]},
'Apple': {'Carl': [(20, 55)], 'Alan': [(80, 80)], 'Dawn': [(40, 80), (40, 85), (-40, 90)]},
'IBM': {'Barb': [(40, 30)]}
}
```

5. Define the summary function, which takes as arguments a **database** (dict) and a dict of current stock prices. It returns a dictionary (dict) whose keys are client names (str) whose associated values are 2-tuples: the first index is a dictionary (dict) whose

keys are stock names (str) and values are the number of shares (int) the client owns. The second index is the amount of money (int) the portfolio is worth (the sum of the number of shares of each stock multiplied by its current price). Assume each client starts with no shares of any stocks, and if a client owns 0 shares of a stock, the stock should not appear in the client's dictionary. Recall positive numbers are a purchase of the stock and negative numbers a sale of the stock. E.g., if db is the **database** above, calling summary(db, {'IBM':65, 'Intel':60, 'Dell':55, 'Apple':70}) would return

{

'Carl': ({'Intel': 20, 'Dell': 20, 'Apple': 20}, 3700), 'Barb': ({'IBM': 40}, 2600),

'Alan': ({'Intel': 20, 'Apple': 80}, 6800),

'Dawn': ({'Apple': 40}, 2800)

}

Some side notes: ***The problems were taken from the ICS33 course taught by Pattis at UCI.***