CS 112 – Fall 2020 – Programming Assignment 9

Sets and Dictionaries

Due Date: Sunday, November 1st, 11:59pm

The purpose of this assignment is to practice operating on new data types: sets and dictionaries.

See the **"assignment basics"** file for more detailed information about getting assistance, running the test file, grading, commenting, and many other extremely important things. Each assignment is governed by the rules in that document.

Guidelines

- You are **not** allowed to `import` anything.
- You are **not** allowed to use anything that hasn't been covered in class, including the *dictionary/list comprehension construct, lambda functions, etc.*
- From built-in functions, you are allowed to call only the following: `range()`, `len()`, `id()`, `set()`, `sum()`, `chr()`, `ord()`
- The `del` operator is allowed anywhere in your code.
- Hard coding is not allowed.
- You should be skimming Piazza regularly in the event that clarifications to this assignment get made.

Testing

You will be provided with a tester to help you check your code, but keep in mind that the grader we use to grade your work will be different from this tester. This means that there will be cases that the tester doesn't check and, therefore, your code might have logic errors that the tester doesn't detect. You must do additional checks, on your own, to make sure that you haven't missed anything and your code is correct. You do ***not*** need to modify the tester we provide, just test on your own any way you like. The goal is to help you put more focus on writing logically correct programs instead of trying to pass certain tests only. Thus, the grader (a.k.a. autograder) will be made available on Gradescope after the deadline.

Grading Rubric

Submitted correctly:	2	# see assignment basics file for file requirements!
Code is well commented:	8	# see assignment basics file for how to comment!
Autograder:	90	
TOTAL:	100	

Note: If your code does not run and crashes due to errors, it will receive **zero** points. Turning in running code is essential.

Assumptions

You may assume that:

- The **types** of the values that are sent to the functions are the proper ones, you don't have to validate them (e.g. `total` will be an integer, not a boolean, etc.).

Functions

The signature of each function is provided below, do ***not*** make any changes to them otherwise the tester will not work properly. The following are the functions you must implement:

`index(d)` **[10pts]**

Description: It takes a dictionary **`d`** that represents a book index with the structure you see below (keys are words that appear in the book and values are lists of page numbers that these words appear in), and returns a new dictionary where the key is the page number and the value is the `set` of words that appear in the respective page.

```
{
   string : [int, int, int, . . . , int],
   string : [int, int, . . . , int],
   . . .
   string : [int, int, int, int, . . . , int],
}
```

Parameters: **`d`** (`dictionary`)

Return value: a new `dictionary`

Examples:

```
d = {'the':[1,3,7,14],'where':[5,7],'after':[3,14]}
index(d)    →   {
                    1:{'the'},
                    3:{'the','after'},
                    5:{'where'},
                    7:{'the','where'},
                    14:{'the','after'}
                }
```

merge(d1,d2) **[20pts]**

Description: It merges two wine inventories represented by dictionaries, **d1** and **d2**, that have the structure depicted below, and returns the merged dictionary.

```
{
    'Chardonnay' : {2017:5, 2018:9, 2019:45, 2020:567},
    'Merlot' : {2020:59, 1988:3, 2019:17, 2018:7},
    . . .
    'Pinot Noir' : {2019:877},
}
```

Parameters: **d1** and **d2** are `dictionaries` – key is a string and value is a dictionary where key and value are `ints`

Return value: a new `dictionary`

Examples:

```
d1 =    {
            'Chardonnay' : {2017:5, 2018:9},
            'Merlot' : {2020:59, 1988:3, 2019:17, 2018:7}
        }
d2 =    {
            'Chardonnay' : {2017:7, 2019:45, 2020:567},
            'Pinot Noir' : {2019:877}
        }
merge(d1,d2)    → {
                    'Chardonnay' : {2017:12, 2018:9, 2019:45, 2020:567},
                    'Merlot' : {2020:59, 1988:3, 2019:17, 2018:7},
                    'Pinot Noir' : {2019:877}
                  }
```

popular_genre(d) **[20pts]**

Description: It takes a dictionary **d** that represents a movie database with the structure you see below, and returns a `set` of movies, the genre of which is the most popular in the database. If two or more genres are equally popular, include all the respective movies in the result. You may **not** use any `lists` or `list` methods.

```
{
    'Inception'      : (2010, {'action', 'adventure', 'sci-fi'}),
    'The Departed'   : (2006, {'crime', 'drama', 'thriller'}),
    'Se7en'          : (1995, {'drama', 'crime', 'mystery'}),
    'Interstellar'   : (2014, {'adventure', 'sci-fi','drama'}),
}
```

Parameters: **d** (`dictionary`) – keys are `strings` and values are `tuples` of (`int`, `set` of `strings`)

Return value: a `set` of movies

Examples:

```
# in the following, the most popular genre is 'drama'
d = {
    'Inception'        : (2010, {'action', 'adventure', 'sci-fi'}),
    'The Departed'     : (2006, {'crime', 'drama', 'thriller'}),
    'Se7en'            : (1995, {'drama', 'crime', 'mystery'}),
    'Interstellar'     : (2014, {'adventure', 'sci-fi','drama'}),
}
popular_genre(d) →    {'The Departed','Se7en','Interstellar'}


# in the following, the most popular genres are 'drama' and 'crime'
d = {
    'Inception'        : (2010, {'action', 'adventure', 'sci-fi'}),
    'The Departed'     : (2006, {'crime', 'drama', 'thriller'}),
    'Se7en'            : (1995, {'drama', 'crime', 'mystery'}),
    'Interstellar'     : (2014, {'adventure', 'sci-fi','drama'}),
    'Ocean\'s Eleven'  : (2001, {'crime', 'thriller'}),
}
popular_genre(d)    →    {'The Departed','Ocean\'s Eleven', 'Se7en','Interstellar'}
```

subset(group, total) **[20pts]**

Description: Creates and returns the smallest subset of **group** that has a sum larger than **total**. If there are more than one subsets of equal size that satisfy the above condition, it selects the one with the largest sum. If there is no subset that satisfies the above condition, it returns `False`. You may **not** use `lists`, `tuples`, `strings`, or `dictionaries`, and you may **not** modify the parameter **group** either.

Parameters: **group** is a `set` of positive `ints` and **total** is a positive `int`

Return value: a new `set`

Examples:

```
subset({6,2,7,3,4,1,5},15)       →       {5,6,7}            # sum=18
# {3,6,7} and {4,6,7} are equally small subsets but their sum is smaller than 18

subset({6,2,7,3,4,1,5},28)       →       False
```

capitalization(d) **[20pts]**

Description: It takes a dictionary **d** that has the structure you see below, and modifies it ***in-place*** by replacing the first character in every string with the respective capital letter. The order of the strings within each list, should remain the same. All strings are guaranteed to contain no characters other than a-z and A-Z. *Hint*: functions `chr()` and `ord()` can be very useful. You may **not** use any string method. From list methods, you may use `.insert()` only.

```
{
   string : [string, string, string, . . . , string],
   string : [string, string, . . . , string],
   . . .
   string : [string, string, string, string, . . . , string],
}
```

Parameters: **d** (`dictionary`)

Return value: Nothing is returned. The modification occurs ***in-place***

Examples:

```
d = {'george':['red','blue','yellow'],'john':['Green','red']}
capitalization(d)       # it doesn't return anything
print(d)    →   {'George':['Red','Blue','Yellow'],'John':['Green','Red']}
```