CSE 101: Introduction to Computational and Algorithmic Thinking

Stony Brook University

Homework Assignment #4

Spring 2018

Assignment Due: April 1, 2018 by 11:59 pm

Assignment Objectives

This homework assignment will give you practice with dictionaries, string processing and file input.

Getting Started

Visit Piazza and download the "bare bones" file `homework4.py` onto your computer, as well as `homework4_driver.py`. Open `homework4.py` in PyCharm and fill in the following information at the top:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)
4. the course number (CSE 101)
5. the assignment name and number (Homework #4)

Do not, under any circumstances, change the names of the functions or their argument lists. The grading software will be looking for exactly those functions provided in `homework4.py`.

Submit your final `homework4.py` file to Blackboard by the due date and time. Late work will not be graded.

Code that crashes and cannot be graded will earn no credit. It is your responsibility to test your code by running it through `homework4.py` and by creating your own test cases.

Part I: Password Strength Calculator (20 points)

Write a function `password_strength()` that takes one argument, `password`, which is a string of characters consisting of lowercase letters, uppercase letters, digits and non-alphanumerical symbols. The function allocates points to `password` based on the location of certain characters in the string:

Rule	Points to Add
the first character is a digit	40
the last character is a digit	50
a digit that is anywhere else	25
the first character is an uppercase letter	25
the last character is an uppercase letter	15
an uppercase letter that is anywhere else	10
a lowercase letter that is anywhere	5
the first character is a symbol	25
the last character is a symbol	35
a symbol that is anywhere else	15

The function returns the total points allocated by the rules above *multiplied by* the length of the password. Double-counting of points *is* acceptable in *some* (not all!) circumstances. For example, consider a string of length 1 that contains an uppercase letter. What is the strength of that string?

For example, if `password` is `'Abce%'`, the total points allocated by the rules above will be 75. Then, 75 $\times$ the length of `password` (5) will give us the return value of 375 total points.

Note: You may assume that the password is non-empty.

Examples:

Function Call	Return Value
`password_strength('f^BAcG')`	`360`
`password_strength('tlX')`	`75`
`password_strength('W63UtHTuN')`	`1170`
`password_strength('f')`	`5`
`password_strength('Msq08#2w&GpMm')`	`2275`
`password_strength('qHjTt9YQ')`	`680`
`password_strength('gw74X5I2')`	`1240`
`password_strength('&9@%B9T(jZJ')`	`1870`
`password_strength('y*n(q%XxVp2')`	`1540`
`password_strength('!fxus')`	`225`

Part II: Order Up! (20 points)

Write a function `order_lunches()` that takes two arguments, in this order:

1. `stock`: A dictionary that contains menu items available for purchase at a restaurant. The dictionary maps items to an integer count of how many of each item is available (e.g., `'soda':4`).
2. `orders`: A list of strings representing items that patrons at the restaurant have ordered. The list might include items not available for purchase.

The function iterates over the `orders` list, treating each string in the list as a key in the `stock` dictionary. If the key is present in the dictionary, the function decrements the value associated with the key (provided that doing so would not make the value negative). The function also returns the total number of valid items purchased. The

function simply ignore strings in `orders` that are not valid keys in `stock`.

As an example, suppose `stock` were the dictionary
`{'soda': 0, 'burger': 3, 'chips': 5, 'pizza': 7}`
and `orders` were
`['soda', 'sandwich', 'burger', 'pizza', 'pizza', 'burger']`.
The function would update `stock` to
`{'soda': 0, 'burger': 1, 'chips': 5, 'pizza': 5}`
and would return `4`.

Note that during grading, different menu items will be used!

Examples:

Function Call	
`order_lunches([{'soda': 8, 'burger': 5, 'chips': 5, 'pizza': 7}, ['pizza', 'pizza', 'burger', 'pizza', 'pizza', 'burger', 'burger']])`	
Return Value	**Updated `stock` Value**
`7`	`{'soda': 8, 'burger': 2, 'chips': 5, 'pizza': 3}`

Function Call	
`order_lunches([{'burger': 4, 'chips': 10, 'pizza': 6, 'soda': 3}, ['soda', 'salad', 'pizza', 'soda', 'chips', 'burger', 'sandwich']])`	
Return Value	**Updated `stock` Value**
`5`	`{'burger': 3, 'chips': 9, 'pizza': 5, 'soda': 1}`

Function Call	
`order_lunches([{'soda': 4, 'chips': 4, 'pizza': 8, 'burger': 9}, ['soda', 'chips', 'chips', 'burger', 'pizza', 'chips', 'soda', 'soup']])`	
Return Value	**Updated `stock` Value**
`7`	`{'soda': 2, 'chips': 1, 'pizza': 7, 'burger': 8}`

Function Call	
`order_lunches([{'chips': 4, 'soda': 6, 'pizza': 6, 'burger': 4}, ['burger', 'pizza', 'burger', 'soup', 'burger', 'soda', 'soda']])`	
Return Value	**Updated `stock` Value**
`6`	`{'chips': 4, 'soda': 4, 'pizza': 5, 'burger': 1}`

Function Call	
`order_lunches([{'soda': 3, 'pizza': 5, 'burger': 6, 'chips': 6}, ['steak', 'chips', 'soda', 'chips', 'burger', 'soda', 'chips', 'chips', 'pizza']])`	
Return Value	**Updated `stock` Value**
`8`	`{'soda': 1, 'pizza': 4, 'burger': 5, 'chips': 2}`

Function Call	
`order_lunches([{'burger': 5, 'soda': 2, 'pizza': 3, 'chips': 1}, ['pizza', 'soda', 'pizza', 'burger', 'burger', 'chips', 'pizza', 'chips', 'soda', 'chips', 'chips', 'burger', 'burger', 'soda']])`	
Return Value	**Updated `stock` Value**
`10`	`{'burger': 1, 'soda': 0, 'pizza': 0, 'chips': 0}`

Function Call	
`order_lunches([{'burger': 3, 'pizza': 4, 'chips': 3, 'soda': 0}, ['steak', 'soda', 'soup', 'chips', 'soda', 'steak', 'soda', 'soda', 'chips', 'soda', 'pizza', 'burger', 'pizza', 'chips', 'chips', 'soda', 'chips']])`	
Return Value	**Updated `stock` Value**
`6`	`{'burger': 2, 'pizza': 2, 'chips': 0, 'soda': 0}`

Function Call	
`order_lunches([{'burger': 4, 'chips': 3, 'pizza': 3, 'soda': 2}, ['salad', 'soda', 'pizza', 'soda', 'burger', 'soda', 'chips', 'soda', 'steak', 'burger', 'soda', 'burger', 'burger', 'burger', 'burger', 'soup']])`	
Return Value	**Updated `stock` Value**
`8`	`{'burger': 0, 'chips': 2, 'pizza': 2, 'soda': 0}`

Function Call	
`order_lunches([{'chips': 5, 'soda': 3, 'burger': 3, 'pizza': 1}, ['soda', 'chips', 'pizza', 'pizza', 'chips', 'salad', 'chips', 'steak', 'soda', 'soda', 'chips', 'burger', 'pizza']])`	
Return Value	**Updated `stock` Value**
`9`	`{'chips': 1, 'soda': 0, 'burger': 2, 'pizza': 0}`

Part III: Population Data (20 points)

Write a function `sum_populations()` that takes five arguments, in this order:

1. `which_continent`: one of the following strings: `'Africa'`,`'Americas'`,`'Asia'`,`'Europe'` or `'Oceania'`
2. `min_gdp`: a positive integer
3. `countries`: a dictionary that maps a continent name to a list of *some* of the countries in that continent
4. `gdps`: a dictionary that maps a country name to that country's GDP (gross domestic product) in millions USD ($MM)
5. `populations`: a dictionary that maps a country name to its population

Here are sample arguments that could be passed to the function:

1. `which_continent = 'Africa'`
2. `min_gdp = 1529760`
3. ```
   countries = {
   'Americas': ['Argentina', 'Uruguay', 'Brazil', 'Puerto Rico', 'Panama'],
   'Oceania': ['Fiji', 'Australia', 'New Zealand', 'Papua New Guinea'],
   'Asia': ['Myanmar', 'Israel'],
   'Africa': ['Senegal', 'Lesotho', 'South Africa', 'Togo', 'Sierra Leone'],
   'Europe': ['Norway', 'Greece']
   }
   ```
4. ```
   gdps = {
   'Argentina': 545866, 'Uruguay': 52420, 'Brazil': 1796186,
   'Puerto Rico': 103135, 'Panama': 55188, 'Fiji': 4632,
   'Australia': 1204616, 'New Zealand': 185017, 'Papua New Guinea': 16929,
   'Myanmar': 67430, 'Israel': 318744, 'Senegal': 14765, 'Lesotho': 2200,
   'South Africa': 294841, 'Togo': 4400, 'Sierra Leone': 3669,
   'Norway': 370557, 'Greece': 194559}
   ```
5. ```
   populations = {'Argentina': 43847430, 'Uruguay': 3444006,
   'Brazil': 207652865, 'Puerto Rico': 3667903, 'Panama': 4034119,
   'Fiji': 898760, 'Australia': 24125848, 'New Zealand': 4660833,
   'Papua New Guinea': 8084991, 'Myanmar': 52885223, 'Israel': 8191828,
   'Senegal': 15411614, 'Lesotho': 2203821, 'South Africa': 56015473,
   'Togo': 7606374, 'Sierra Leone': 7396190, 'Norway': 5254694,
   'Greece': 11183716}
   ```

Using the given arguments, the function computes and returns the sum of populations of all countries that are located in `continent` and which have a GDP that is greater than or equal to `min_gdp`.

For example, suppose `which_continent = 'Americas'` and `min_gdp = 1000000`. The function would consult the `countries` dictionary to get a list of countries from the Americas. It would then consult the `gdps` dictionary to find all countries with a GDP of at least $1,000,000M and add together the populations of all such countries.
   ```

Examples:

Due to the large amount of space consumed by examples, only two are provided here. See the driver file for more test cases.

Function Call	Return Value
sum_populations('Africa', 42690, {'Americas': ['Panama', 'Guatemala'], 'Oceania': ['New Zealand', 'Australia', 'Fiji'], 'Asia': ['Vietnam', 'South Korea', 'Israel'], 'Africa': ['Lesotho', 'Burundi', 'Algeria', 'Cameroon', 'Angola'], 'Europe': ['Finland', 'Luxembourg', 'Russia', 'Belarus', 'Ukraine']}, {'Panama': 55188, 'Guatemala': 68763, 'New Zealand': 185017, 'Australia': 1204616, 'Fiji': 4632, 'Vietnam': 202616, 'South Korea': 1411246, 'Israel': 318744, 'Lesotho': 2200, 'Burundi': 3007, 'Algeria': 156080, 'Cameroon': 24204, 'Angola': 89633, 'Finland': 236785, 'Luxembourg': 59948, 'Russia': 1283162, 'Belarus': 47433, 'Ukraine': 93270}, {'Panama': 4034119, 'Guatemala': 16582469, 'New Zealand': 4660833, 'Australia': 24125848, 'Fiji': 898760, 'Vietnam': 94569072, 'South Korea': 50791919, 'Israel': 8191828, 'Lesotho': 2203821, 'Burundi': 10524117, 'Algeria': 40606052, 'Cameroon': 23439189, 'Angola': 28813463, 'Finland': 5503132, 'Luxembourg': 575747, 'Russia': 143964513, 'Belarus': 9480042, 'Ukraine': 44438625})	69419515
sum_populations('Oceania', 386428, {'Americas': ['Peru', 'Dominican Republic', 'United States'], 'Oceania': ['New Zealand', 'Australia'], 'Asia': ['India', 'Jordan', 'Tajikistan'], 'Africa': ['Benin', 'Central African Republic', 'South Africa', 'Namibia', 'Rwanda'], 'Europe': ['Germany', 'Luxembourg', 'Russia']}, {'Peru': 192094, 'Dominican Republic': 71584, 'United States': 18624475, 'New Zealand': 185017, 'Australia': 1204616, 'India': 2263792, 'Jordan': 38655, 'Tajikistan': 6952, 'Benin': 8583, 'Central African Republic': 1756, 'South Africa': 294841, 'Namibia': 10267, 'Rwanda': 8376, 'Germany': 3477796, 'Luxembourg': 59948, 'Russia': 1283162}, {'Peru': 31773839, 'Dominican Republic': 10648791, 'United States': 322179605, 'New Zealand': 4660833, 'Australia': 24125848, 'India': 1324171354, 'Jordan': 9455802, 'Tajikistan': 8734951, 'Benin': 10872298, 'Central African Republic': 4594621, 'South Africa': 56015473, 'Namibia': 2479713, 'Rwanda': 11917508, 'Germany': 81914672, 'Luxembourg': 575747, 'Russia': 143964513})	24125848

Part IV: Assembling a Course Roster (20 points)

In this part you will write a function that opens a file containing a list of students enrolled in courses and extract some data from the file. Each line of the file (after the first line) contains the name of a student, followed by the student's ID number, followed by a course code, followed by the number of credits for the course. These four quantities are separated by commas. The first line always contains the string `'Name,StudentID,CourseCode,Credits'`.

Complete the function `get_roster()` that takes the following arguments, in this order:

1. `filename`: The name of a file that the function will read data from. You may assume that the file is always validly formatted.
2. `course`: A string representing a course code we are interested in.

The function reads each line of the file `filename` and returns the list of ID numbers of students enrolled in the designated `course`. You may assume that each line of data in the file will always follow the following format: (Student's Name, Student's ID #, Course Code, # of Credits). Note that ALL values in the file are treated as strings, even values we would normally treat as numerical. Below is a portion of a sample input file:

```
Name,StudentID,CourseCode,Credits
Ariel,110071434,CSE101,3
Aldo,110071435,CSE220,3
Julio,110071432,CSE220,3
Aldo,110071435,CSE114,4
Natalie,110071433,CSE214,3
Ariel,110071434,CSE219,4
Destiny,110071436,CSE214,3
```

Note: When reading each line, make sure to use the `strip()` function to get rid of any newlines and/or spaces before or after each line.

Examples:

See Piazza for the contents of `students1.txt`, `students2.txt` and `students3.txt`. Different files *will* be used during grading.

Function Call	**Return Value**
`get_roster('students1.txt', 'CSE220')`	`['110071435', '110071432', '110071434']`
`get_roster('students1.txt', 'CSE114')`	`['110071435', '110071434']`
`get_roster('students1.txt', 'CSE475')`	`['110071436', '110071435']`
`get_roster('students1.txt', 'CSE219')`	`['110071434', '110071432', '110071433']`
`get_roster('students1.txt', 'CSE214')`	`['110071433', '110071436']`

`get_roster('students2.txt', 'CSE219')`	`['110071438']`
`get_roster('students2.txt', 'CSE214')`	`['110071447', '110071450', '110071441', '110071442']`
`get_roster('students2.txt', 'CSE220')`	`['110071446', '110071449', '110071448']`
`get_roster('students2.txt', 'CSE114')`	`['110071449', '110071447', '110071444', '110071441', '110071448', '110071437']`
`get_roster('students2.txt', 'CSE101')`	`['110071437', '110071446']`
`get_roster('students3.txt', 'CSE220')`	`['110071434', '110071441', '110071438', '110071449', '110071439', '110071445', '110071435']`
`get_roster('students3.txt', 'CSE101')`	`['110071442', '110071438', '110071435', '110071448', '110071432']`
`get_roster('students3.txt', 'CSE219')`	`['110071435', '110071442', '110071446', '110071438', '110071437']`
`get_roster('students3.txt', 'CSE114')`	`['110071451', '110071441', '110071437', '110071435']`
`get_roster('students3.txt', 'CSE214')`	`['110071449', '110071440', '110071436']`

Part V: Fetch Information (20 points)

In this part we will generalize the results of the previous part to files that contain information other than course enrollment data. The first line of the file will contain a comma-separated list of strings that define the format of the file. Some examples:

- Car data: `Make,Model,Year`
- CPU data: `Manufacturer,Model,NumCores,ClockSpeed`
- House data: `Town,Year,Price,NumRooms,Taxes`

Write a function `fetch_value()` that takes the following arguments, in this order:

1. `filename`: The name of a file the function will read data from. All fields in the file will be separated by commas. You may assume that the file is always validly formatted.
2. `selected_field`: The name of the field from which we are *reading* values.
3. `searched_field`: The name of the field we are using to *search* the data.
4. `searched_value`: The value of the `searched_field` that we are trying to *match*.

To understand the meaning of these arguments, suppose the function were called on CPU data (with fields `Manufacturer,Model,NumCores,ClockSpeed`) with the following arguments:

`fetch_value('cpu.txt', 'Manufacturer', 'ClockSpeed', '3')`. This function call indicates that we want a list of CPU manufacturers whose CPU's `ClockSpeed` field has a value of `'3'`. (Duplicated values are expected, and the number of copies of a selected value must equal the number of lines that match the `search_value`. ***Do not sort the returned list.***) Note that ALL values in the file are treated as strings, even values we would normally treat as numerical.

More generally speaking, the function reads from the given file and finds lines where the value in the `searched_field` "column" matches the `searched_value` function argument. When such a line is found, the function appends the corresponding value of `selected_field` column into the returned list.

Examples:

See Piazza for the contents of `clothing.txt` and `sundaes.txt`. Different files *will* be used during grading.

Function Call

```
fetch_value('sundaes.txt', 'Topping', 'Flavor', 'RockyRoad')
```

Return Value

```
['WhippedCream', 'WhippedCream', 'ChocolateChips', 'ChocolateChips',
'Sprinkles', 'WhippedCream', 'ChocolateChips', 'ChocolateChips',
'Sprinkles', 'Sprinkles', 'ChocolateChips']
```

Function Call

```
fetch_value('sundaes.txt', 'Flavor', 'Size', 'Medium')
```

Return Value

```
['Strawberry', 'RockyRoad', 'Strawberry', 'Vanilla', 'Strawberry',
'RockyRoad', 'Strawberry', 'Chocolate', 'Chocolate', 'Chocolate',
'Chocolate', 'RockyRoad', 'Strawberry', 'Vanilla', 'Chocolate',
'Vanilla']
```

Function Call

```
fetch_value('sundaes.txt', 'Syrup', 'Topping', 'WhippedCream')
```

Return Value

```
['Fudge', 'Fudge', 'Fudge', 'Fudge', 'Blueberry', 'Blueberry',
'Blueberry', 'Fudge', 'Blueberry', 'Caramel', 'Fudge', 'Caramel',
'Blueberry', 'Fudge', 'Caramel']
```

Function Call

```
fetch_value('sundaes.txt', 'Topping', 'Syrup', 'Fudge')
```

Return Value

```
['WhippedCream', 'WhippedCream', 'WhippedCream', 'WhippedCream',
'ChocolateChips', 'ChocolateChips', 'Sprinkles', 'Sprinkles',
'ChocolateChips', 'Sprinkles', 'WhippedCream', 'WhippedCream',
'ChocolateChips', 'WhippedCream', 'ChocolateChips']
```

Function Call
`fetch_value('clothing.txt', 'BodyType', 'BodyType', 'Girl')`
Return Value
`['Girl', 'Girl', 'Girl', 'Girl', 'Girl', 'Girl', 'Girl']`

Function Call
`fetch_value('clothing.txt', 'ClothingType', 'BodyType', 'Woman')`
Return Value
`['Pants', 'Socks', 'Coat', 'Pants', 'Coat', 'Coat', 'Pants']`

Function Call
`fetch_value('clothing.txt', 'Manufacturer', 'Size', 'Small')`
Return Value
`['Chanel', 'Aeropostale', 'Lee', 'Lee', 'Adidas', 'Levi', 'Lee', 'Lee', 'Aeropostale', 'Levi', 'Lee', 'Levi']`

Function Call
`fetch_value('clothing.txt', 'Price', 'Size', 'XL')`
Return Value
`['24', '42', '60', '60', '24']`

How to Submit Your Work for Grading

To submit your `.py` file for grading:

1. Login to Blackboard and locate the course account for CSE 101.
2. Click on "Assignments" in the left-hand menu and find the link for this assignment.
3. Click on the link for this assignment.
4. Click the "Browse My Computer" button and locate the `.py` file you wish to submit. Submit only that one `.py` file.
5. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the above directions again. We will grade only your last submission.