CSE 101: Introduction to Computational and Algorithmic Thinking

Stony Brook University

Homework Assignment #2

Spring 2018

Assignment Due: February 16, 2018 by 11:59 pm

Assignment Objectives

This homework assignment will give you practices on if-statements, lists and for-loops.

Getting Started

Visit Piazza and download the "bare bones" file `homework2.py` onto your computer, as well as `homework2_driver.py`. Open `homework2.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 #2)

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 `homework2.py`.

Submit your final `homework2.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 `homework2.py` and by creating your own test cases.

Part I: Compute a Summation (20 points)

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

1. `x`: the first value needed for a formula
2. `y`: the second value needed for a formula

The function computes and returns the value of the formula given below (using floating-point division):

$$\sum_{k=x}^{y}\left[(k^3-1)+\frac{5k-10}{3}\right]$$

Examples:

Function Call	Return Value
`summation(-9, 1)`	`-2145.0`
`summation(5, 9)`	`1961.6666666666665`
`summation(-5, -3)`	`-249.0`
`summation(-7, -2)`	`-854.0`
`summation(6, 9)`	`1832.6666666666665`
`summation(0, 10)`	`3069.0`
`summation(-4, 8)`	`1183.0`
`summation(2, 3)`	`34.66666666666667`
`summation(-6, 9)`	`1554.6666666666665`
`summation(-9, 4)`	`-2044.0000000000002`

Part II: Gas Reward Card (20 points)

Imagine a gas station that offers a rewards card program. When a customer buys gas, he or she is rewarded with rewards points that can be used to purchase other goods.

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

1. `current_points`: the current rewards points (an integer) on the customer's gas rewards card
2. `gas_type`: the type of gasoline purchased (a string), which is one of the following:
   - `'Regular'`: $2.65 / gallon
   - `'Plus'`: $2.90 / gallon
   - `'Premium'`: $3.10 / gallon
3. `money_spent`: the amount of dollars spent on gas for a particular transaction. This number is needed to compute how many gallons of gas were purchased.

As previously mentioned, after each gas refill, the customer gets a certain amount of rewards points. The function calculates and returns the *updated*, *new total* rewards points of the customer. The rewards rules are as follows:

1. Normally, 5 points are rewarded for every *full dollar* spent. For example, $6.83 spent will result in $5 \times 6 = 30$ new reward points. Hint: given a floating-point variable `x`, the code `int(x)` will *truncate* `x`, thereby giving only the integer part of `x`: `integer_part = int(x).`
2. If *at least* 10 gallons of gas were purchased, the customer will be rewarded 200 reward points
3. If *at least* 15 gallons of gas were purchased, the customer will be rewarded 400 reward points
4. If *at least* 20 gallons of gas were purchased, the customer will be rewarded 550 reward points

Note: Only one of the above rules can be applied per transaction. For example, if a customer bought 16 gallons of gas, he/she will get 400 and only 400 reward points. Rule 1 is applied for purchases of less than 10 gallons.

Special Cases: Below are some special cases that the function must handle. Your code should check for these special cases *in this order*:

- If `money_spent` is less than one dollar, this is considered an invalid transaction, and the function returns `current_points`.
- If `current_points` is less than 0, this is considered a faulty account, and the function returns `'error'` (a string).
- If `gas_type` is not one of the three kinds provided, this means it's not available, and the function returns `'unavailable'` (a string).

Examples:

Function Call	**Return Value**
`gas_reward(1995, 'Regular', 74.95480578251386)`	`2545`
`gas_reward(870, 'Regular', -23.705557429821557)`	`870`
`gas_reward(3054, 'Good', 27.80750330446164)`	`'unavailable'`
`gas_reward(4933, 'Plus', 76.85532456142764)`	`5483`
`gas_reward(393, 'Regular', -7.245453064858392)`	`393`
`gas_reward(1981, 'Plus', 5.838288853611488)`	`2006`
`gas_reward(4989, 'Regular', -29.623101627776485)`	`4989`
`gas_reward(3033, 'Premium', 54.08601795248893)`	`3433`
`gas_reward(1792, 'Plus', 25.587222964208976)`	`1917`
`gas_reward(4666, 'Regular', 57.73657576959238)`	`5216`
`gas_reward(-5, 'Plus', 58.933838226988605)`	`'error'`

Note: The quotation marks displayed in the example return values are there to emphasize that the return values are strings. You should not add quotation marks to your return values.

Part III: Letter Puller (20 points)

Write a function `pull()` that takes two arguments, in the following order:

1. `strings`: a list of strings (e.g., `['abcde', 'fghj', 'jk]`)
2. `i`: an integer that indicates which character of each string the function will extract (pull out)

The function extracts the character at index `i` of each string, appends the characters to an empty list, and then returns the resulting list.

For example, if `i=2`, then the function extracts the character at index `i` of each string, *if such a character exists*, and appends them to a list. For the list of strings above the returned result would be `['c', 'h']`.

Hints:

- Recall that the valid indexes of a list are `0` through `len(list_name)-1`. Use this fact to see if a list is long enough to have a character with index `i`.
- To create an empty list, type `list_name = []`, where `list_name` is the name of the list.
- Use the `append` method to append an item to the end of a list. For example, to append `letter` to the list called `result`, type `result.append(letter)`.

Examples:

Function Call	Return Value
`pull(['Z9', 'gbd2xKC', '7MWt0', 'dcl2'], 1)`	`['9', 'b', 'M', 'c']`
`pull(['GMca', 'xuP5S'], 1)`	`['M', 'u']`
`pull(['p8K5m', 'OvnLHqz'], 6)`	`['z']`
`pull(['QGqj3ZM', 'ZD', 'fU2eN81n', 'RztBk', 'ySTuI1', 'FHLoDVw0', 'WUybIBOA'], 1)`	`['G', 'D', 'U', 'z', 'S', 'H', 'U']`
`pull(['vFmq', '5wVxmCT'], 6)`	`['T']`
`pull(['EXdfk', 'T2RCPQn5'], 0)`	`['E', 'T']`
`pull(['Q7A', '4Dhm8G1j', 'BQ1FH'], 0)`	`['Q', '4', 'B']`
`pull('YFnOfD', 'o2eMrAY', 'EofzJ', 'OxEot13z', 'jmksbi'], 5)`	`['D', 'A', '1', 'i']`
`pull(['yCZMkI', '2i'], 2)`	`['Z']`
`pull(['iL', 'C8n7', 'vj5gq', 'n1', 'NJ20EYMx', 'D9ylhBQ', 'mr15iUX', 'zS'], 4)`	`['q', 'E', 'h', 'i']`

Note: The quotation marks displayed in the example return values are there to emphasize that the return values are strings. You should not add quotation marks to your return values.

Part IV: Print Job Cost Calculator (20 points)

Write a function `print_cost()` that takes one argument, `task`, which is a list of three elements:

1. The first element is a string that represents the size of the paper used:

Paper Size (string)	Cost per Sheet ($)
'Letter'	0.05
'Legal'	0.06
'A4'	0.055
'A5'	0.04

2. The second element is a string that represents the color of the printing. This cost is added to the cost of the paper itself:

Printing Type (string)	Cost per Sheet ($)
'Gray Scale'	0.01
'Colored'	0.10

3. The third element is an integer that represents the quantity of paper used for the print job.

The function computes and returns the total cost of the current printing job.

Note: You may assume that all function arguments will be valid (e.g., no negative values, invalid paper sizes, etc.)

Examples:

Function Call	**Return Value**
`print_cost(['Legal', 'Gray Scale', 44])`	`3.0799999999999996`
`print_cost(['Letter', 'Gray Scale', 4])`	`0.24000000000000002`
`print_cost(['A4', 'Colored', 32])`	`4.96`
`print_cost(['A4', 'Colored', 4])`	`0.62`
`print_cost(['Letter', 'Gray Scale', 16])`	`0.9600000000000001`
`print_cost(['Letter', 'Colored', 33])`	`4.950000000000001`
`print_cost(['Letter', 'Gray Scale', 32])`	`1.9200000000000002`
`print_cost(['A4', 'Colored', 42])`	`6.51`
`print_cost(['Legal', 'Gray Scale', 28])`	`1.9599999999999997`
`print_cost(['Legal', 'Gray Scale', 44])`	`3.0799999999999996`

Part V: Total Printing Cost (20 points)

This part is a continuation of Part IV. However, instead of receiving a single print job (a single list), you will be given many print jobs (a list of lists, in which each list is a print job).

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

1. `ID`: an integer that represents a student's ID number
2. `tasks`: is a *list of lists* that contains individual *sub-lists* as described in Part IV. However, each sub-list has one more element added to the front of it, which is `ID`. The ID number is the first element of each sub-list.

   An example of this is: `[[102423, 'Legal', 'Gray Scale', 44], [142442, 'Letter', 'Gray Scale', 4], [514709, 'A4', 'Colored', 32]]`

   This list represents a list of three print jobs for students with ID numbers 102423, 142442 and 514709, respectively.

The function looks through the list of print jobs, finds all print job that belong to the student whose ID# is `ID`, and returns the total cost of those print jobs.

Note: You may assume that all function arguments will be valid

Hint: You might want to reuse the code you wrote in Part IV by calling your `print_cost()` function. Doing this will greatly reduce the amount of code you will need to write for Part V.

Examples:

Function Call	**Return Value**
`total_cost(6, [[6, ’A5’, ’Gray Scale’, 25], [2, ’Legal’, ’Colored’, 33], [7, ’A4’, ’Colored’, 15], [2, ’A5’, ’Colored’, 39], [4, ’A4’, ’Gray Scale’, 47], [5, ’A4’, ’Colored’, 13], [8, ’Letter’, ’Gray Scale’, 33], [6, ’Legal’, ’Gray Scale’, 17], [8, ’Letter’, ’Colored’, 39]])`	`2.44`
`total_cost(3, [[8, ’Legal’, ’Gray Scale’, 30], [3, ’A5’, ’Colored’, 43], [8, ’Letter’, ’Colored’, 31]])`	`6.0200000000000005`
`total_cost(6, [[2, ’Legal’, ’Colored’, 33], [6, ’A4’, ’Gray Scale’, 5], [6, ’Legal’, ’Colored’, 26]])`	`4.485`
`total_cost(8, [[8, ’Legal’, ’Colored’, 48]])`	`7.68`
`total_cost(8, [[8, ’Letter’, ’Gray Scale’, 33], [1, ’Legal’, ’Colored’, 39], [4, ’A4’, ’Gray Scale’, 47], [8, ’A5’, ’Colored’, 36], [8, ’Letter’, ’Colored’, 39], [8, ’Letter’, ’Gray Scale’, 38], [4, ’A5’, ’Gray Scale’, 35], [5, ’A5’, ’Gray Scale’, 13], [8, ’Letter’, ’Colored’, 31]])`	`19.800000000000004`
`total_cost(8, [[8, ’Letter’, ’Gray Scale’, 38], [8, ’Legal’, ’Gray Scale’, 30], [8, ’Legal’, ’Gray Scale’, 33], [8, ’A5’, ’Colored’, 36], [8, ’Legal’, ’Colored’, 48], [7, ’A5’, ’Gray Scale’, 6)`	`19.41`
`total_cost(8, [[8, ’Letter’, ’Colored’, 31], [8, ’Legal’, ’Gray Scale’, 33], [6, ’A4’, ’Gray Scale’, 5], [4, ’A4’, ’Gray Scale’, 47], [2, ’A5’, ’Colored’, 39], [8, ’Letter’, ’Colored’, 39], [7, ’A4’, ’Gray Scale’, 39], [7, ’Legal’, ’Colored’, 3], [7, ’A5’, ’Gray Scale’, 8]])`	`12.81`

`total_cost(3, [[10, 'A4', 'Colored', 4],`
`[9, 'A4', 'Colored', 30],`
`[4, 'A5', 'Colored', 41],`
`[3, 'A5', 'Colored', 43],`
`[1, 'A5', 'Gray Scale', 47],`
`[8, 'Legal', 'Colored', 48],`
`[1, 'Letter', 'Gray Scale', 3],`
`[6, 'A5', 'Gray Scale', 35],`
`[8, 'Letter', 'Gray Scale', 38],`
`[6, 'A4', 'Colored', 13]])`
	`6.0200000000000005`

`total_cost(8, [[6, 'A5', 'Gray Scale', 35],`
`[8, 'Legal', 'Colored', 39],`
`[7, 'A4', 'Colored', 44],`
`[10, 'A4', 'Colored', 4],`
`[1, 'Letter', 'Gray Scale', 3],`
`[7, 'A5', 'Gray Scale', 8],`
`[1, 'Legal', 'Colored', 39],`
`[7, 'Legal', 'Colored', 3],`
`[3, 'A5', 'Colored', 43],`
`[4, 'Letter', 'Colored', 27]])`
	`6.24`

`total_cost(9, [[9, 'A4', 'Colored', 30]])`	`4.65`

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.