COL100 Assignment 6
2nd Semester Semester : 2021-2022

Deadline: 11:59 pm, 8 May, 2022

General Instructions

You should attempt this assignment without taking help from your peers or referring to online resources except for documentation (we will perform a **plagiarism check** amongst all submissions). Any violation of above will be considered a breach of the honor code, and the consequences would range from **zero marks** in the assignment to a **disciplinary committee action**.

Submission Instructions

1. If you are solving the i^{th} question, code in a file named `<EntryNo>-qi.py`. For eg. solution code for the 2^{nd} question goes inside a file named `<EntryNo>-q2.py`.
2. You are **not** required to submit the lab question mentioned in the assignement as "In Lab Component". The question evaluation for a lab will be done in the lab only and evaluations not done for the in-lab component for a student in his/her lab slot will be marked 0. It is your duty to get them evaluated.
3. Submit your code in a `.zip` file named in the format `<EntryNo>.zip`. Make sure that when we run unzip `<EntryNo>.zip`, a folder `<EntryNo>` should be produced in the current working directory. For eg. if your entry number is `2021CS5XXXX`, then your zip file would be `2021CS5XXXX.zip` and upon unzipping, it should produce a folder `2021CS5XXXX` containing files `<EntryNo>-q1.py`, `<EntryNo>-q2.py`, `<EntryNo>-q3.py` and so on. For reference, we would be uploading a sample zip file on moodle containing the exact directory structure which is mentioned.
4. Your submissions will be **auto-graded**. Make sure that your code follows the specifications (including directory structure, input/output, importing libraries, submission `.zip` file) of the assignment precisely.

Some Clarifications

1. Every problem description is followed by some examples showing how exactly input and output is being expected. Please refer to them for more clarity.
2. You must not change the parameters and the names of the functions provided in the skeleton code unless clearly specified.
3. All the inputs/outputs are **integers** unless explicitly mentioned.
4. If you still have any more doubts, feel free to shoot them at Piazza.
5. You can only use lists or arrays to store the data in the questions.
6. Remember to **not to import** anything and you can use common inbuilt functions like append/split/len/etc freely.

1 Lunch Break (10 marks)(In Lab Component)

Consider the arrangement of plates in your mess. You have to write a program to let students have their lunch properly. The plate which is at the top is the first one to be picked by a student. When you add a plate you always add to the top of the arrangement of the plates. Each plate has a unique integer number, call it the plate ID, assigned. Note that this is not related to the position/order or the arrangement. As students come to lunch they pick up plates for lunch.

In this question you will need to implement three operations - `AddPlate`, `PickPlate` and `Check`. `AddPlate` add a new plate with given plate ID on the top of arrangement ,`PickPlate` removes the top most plate from the arrangement and `Check` returns the top most plate ID in the arrangement.

You may implement these as functions named `AddPlate()`, `PickPlate()` and `Check()` for the operations `AddPlate`, `PickPlate` and `Check` respectively. You don't need to worry about the efficiency of these operations/functions. We have provided you with a basic skeleton code that you have to fill, please download that from Gradescope and fill in the functions. You can implement them however you want. The aim of providing it is to make sure you use functions and make them a habit. You can't change the name of the function and parameters of a function. The code has comments which will be explaining it well.

The only time you will be printing Plate ID is when the `Check` is called and you will also print when there is an invalid operation. When you try to `Check` or `PickPlate` when no plate is present, print `Invalid`.

The first line of input is the amount of queries to be processed. Invalid, which is to be printed, is case sensitive.

NOTE: Queries like "AddPlate 5" and "Check" are given as string input and you have to call the function with appropriate parameters (if any) from these input strings

NOTE: Each Queries will be in new line. Please be careful with indentation and case sensitivities

NOTE: Only print integers. No need to use $\%.2f$

NOTE: In this question we will be checking ONLY your output so make sure it follows the correct format. Your functions returned value will not be checked. But if functions are not used marks will be deducted.

Example 1:
`INPUT:`

```
7
AddPlate 5
Check
AddPlate 4
Check
AddPlate 3
PickPlate
Check
```

`OUTPUT:`

```
5
4
4
```

`EXPLANATION:`

- Initially the line is empty.

- The first operation is `AddPlate` 5. So we add a plate with ID 5 into the line.
- Next operation called is `Check` so **Check** prints the plate at the top of the arrangement which is 5.
- Then we **AddPlate** 4 as well thus the arrangement now have plates $5, 4$ with 4 at the top.
- **Check** prints 4 which is currently at the top of the arrangement.
- We **AddPlate** 3 to the line making it $5, 4, 3$ with 3 at the top. The line currently is $5, 4, 3$.
- We then **PickPlate** 3 a ticket as it is on the top of the arrangement -we are left with $5, 4$.
- Check prints 4.

Example 2:
`INPUT`:

```
8
AddPlate 3
AddPlate 1
Check
PickPlate
Check
PickPlate
Check
PickPlate
```

`OUTPUT`:

```
1
3
Invalid
Invalid
```

`EXPLANATION`:
We add Plate 3 and 1 making it $\{3, 1\}$ then for Check we receive 1. Then We pick the top most plate which is 1 and we are left with 3 thus check prints 3. We pick the last plate and now there are no plates left so the operations - Check and PickPlate are "Invalid".

Example 3:
`INPUT`:

```
9
AddPlate 53
AddPlate 67
Check
PickPlate
AddPlate 72
Check
PickPlate
PickPlate
PickPlate
```

`OUTPUT`:

```
67
72
Invalid
```

2 Movie Time (10 marks)

Consider people in a line to buy a movie ticket. You are the ticket booth operator and want to create a program to give tickets to the people in the line. Each person has a unique integer,which is also called Aadhar ID (It can be any integer and not necessarily original 12 digit ID) , assigned and is used to identify the person. As people come you keep on issuing tickets to them. The person at the beginning of the line is the first one to receive the movie ticket, therefore a person who arrived earlier will get their ticket first and then that person leaves the line. One person will only take one ticket. Initially the line is empty.

In this question you will be doing four operations - `Join`, `Issue`, `GetLine` and `Check`. `Join` adds a person into the line of people in the end of the line, `Issue` issues a ticket to the front most person and that person leaves the line and `Check` gives the Aadhar ID of the person at the beginning of the line to check for credentials. `GetLine` will give us the Aadhar ID of all the people in the line starting from the first person to the person in the end (in this order) at that instant **separated by a single space**.

You have to implement these as functions named `Join()`, `Issue()`, `GetLine()` and `Check()` for the operations `Join`, `Issue`, `GetLine` and `Check` respectively. You don't need to worry about the efficiency of these operations/functions. We have provided you with a basic skeleton code that you have to fill, please download that from Gradescope and fill in the functions. You can implement them however you want. The aim of providing it is to make sure you use functions and make them a habit. You can't change the name of the function and parameters of a function. The code has comments which will be explaining it well.

You will be printing the ID when `Check` and `GetLine` is called. If there is no person in the line and `Check`, `Issue` or `GetLine` is called, print `Invalid`.

Input: The first line of input is the number of operations, call it n, that will be given. The next n lines will contain the operation. When `Join` will be called there would be a number followed which will be the ID of the person which joins at the end of the line.

Output: Only print the output asked (no prompts or error code is to be printed). Print the Aadhar ID when the operations `GetLine` and `Check` are given or `Invalid` when there is a invalid operation called. Only print integers. Invalid, which is to be printed, is case sensitive

NOTE: Queries like "Join 5" ; "Check" are given as string input and you have to call the function with appropriate parameters (if any) from these input strings

NOTE: Each Queries will be in new line. Please be careful with indentation and case sensitivities

NOTE: Only print integers. No need to use $\%.2f$

NOTE: In this question we will be checking ONLY your output so make sure it follows the correct format. Your functions returned value will not be checked. But if functions are not used marks will be deducted.

Example 1:
`INPUT`:

```
10
Join 5
Check
```

```
Join 4
Check
GetLine
Join 3
GetLine
Issue
Check
GetLine
```

OUTPUT:

```
5
5
5 4
5 4 3
4
4 3
```

EXPLANATION:

- Initially the line is empty.
- The first query is `Join` 5. So we **Join** person with ID 5 into the line.
- Next operation called is `Check` so **Check** prints the person in the beginning of the line which is 5.
- Then we **Join** 4 as well thus the line now have people $5, 4$ with 5 at front.
- **Check** prints 5.
- **GetLine** prints 5 4.
- We **join** 3 to the line making it $5, 4, 3$ with 5 at front. The line currently is 5 4 3.
- We then **issue** 5 a ticket as he is in the beginning of the line - 5 will take the ticket and will leave the line, we are left with $4, 3$.
- **Check** prints 4 as 4 is currently in the beginning of the line.
- **GetLine** prints the current line 4 3

Example 2:
INPUT:

```
11
Join 1
Join 3
GetLine
Check
Issue
GetLine
Check
Issue
Getline
Check
Issue
```

OUTPUT:

```
1 3
1
3
3
Invalid
Invalid
Invalid
```

`EXPLANATION`:
We `Join` 1 and 3 into the line making it $1, 3$ with 1 at the front. Then `Getline` prints 1 3. When we do a `Check`, we receive 1. We issue the front most person which is 1 and we are left with 3 thus `Getline` prints 3 and `Check` prints 3. We issue 3 a ticket and now the line is empty and the operations - `GetLine`, `Check` and `Issue` are `Invalid`.

Example 3:
`INPUT`:

```
9
Join 53
Join 67
Check
Issue
Join 72
Check
Issue
Issue
Issue
```

`OUTPUT`:

```
53
67
Invalid
```

`EXPLANATION`:
Our line has $53, 67$ so we check the person with ID 53. We issue 53 with a ticket and are left with 67. 72 joins the line making our line $67, 72$. We check the person with ID 67. We issue 67 and 72 but cannot issue anymore tickets and print **Invalid**.

3 The weird restaurant - Smart or fool? (15 marks)

In a town, there is a restaurant which is famous because it has a weird method of operation. It provides its customer an ID (which is unique and has nothing to do with the arrival order of the customers) and the bill amount to be paid. The restaurant thinks that by giving privilege to the customer with higher bill amount they can provoke the customers to pay more so that they get their order faster. Hence the restaurant will serve the customer first which has the highest bill amount in the waiting line. If the bill amount is same then the customer which came first will be served first.

In this question, you will be implementing three operations - `Order`, `Serve` and `Highest`. `Order` inserts the customer into the waiting line, `Serve` serves the highest bill customer and thus the customer will leave the waiting line which has the highest bill amount, `Highest` gives the customer ID number having the highest bill amount in the waiting line at the moment. Note that if the bill amount is same, the customer ID that was inserted first in the chronological order is considered.

Input: First line contains an integer which indicates the number of operations to be performed. From next line on wards the operations are followed. When calling `Order`, we provide two space separated numbers, first number is the customer ID to be inserted, and the second number is the bill amount for the customer. Refer to the examples below for more clarity.

Output: You will be printing the customer ID only when `Highest` is called. If you call **Highest** or **Serve**, when no element is present, print `Invalid`. Also note that There may be multiple orders with same bill amount inserted.

You may implement these as functions named `Order()`, `Serve()` and `Highest()` for the operations `Order`, `Serve` and `Highest` respectively. You don't need to worry about the efficiency of these operations/functions. We have provided you with a basic skeleton code that you have to fill, please download that from Gradescope and fill in the functions. You can implement them however you want. The aim of providing it is to make sure you use functions and make them a habit. You can't change the name of the function and parameters of a function. The code has comments which will be explaining it well.

NOTE: Queries like "Order 5 20" ; "Highest" are given as string input and you have to call the function with appropriate parameters (if any) from these input strings

NOTE: Each Queries will be in new line. Please be careful with indentation and case sensitivities

NOTE: Only print integers. No need to use $\%.2f$

NOTE: In this question we will be checking ONLY your output so make sure it follows the correct format. Your functions returned value will not be checked. But if functions are not used marks will be deducted.

Example 1:
`INPUT`:

```
8
Order 5 20
Highest
Order 4 100
Highest
Order 3 100
Highest
Serve
Highest
```

`OUTPUT`:

```
5
4
4
3
```

`EXPLANATION`:
When we insert 5 into the waiting line having bill amount as 20. Calling Highest would print 5. We insert 4 having bill amount as 100. Calling highest now would print 4 as customer with ID 4 has highest bill amount till now. We then insert 3 having bill amount as 100. Calling Highest would still print 4 because even though 4 and 3 have the same bill amount, 4 would be considered since it was inserted before 3. Remove now removes 4 from the waiting line. Highest now prints 3 as the customer with ID 3 has higher bill amount than customer with ID 5.

Example 2:
`INPUT`:

```
8
Order 1 22
Order 7 23
Highest
```

```
Serve
Highest
Serve
Highest
Serve
```

OUTPUT:

```
7
1
Invalid
Invalid
```

EXPLANATION:
When we insert 1 into the waiting line with bill 22, then we insert 7 into waiting line with bill 23. We get the highest order which is 7 with bill amount 23. Then we serve the highest order leaving only order 1 in the waiting line. Highest gives us order 1 as it's the only one left. We serve the order 1 and are left with an empty waiting line. The operations `Highest` and `Serve` are `Invalid`.

Example 3:
INPUT:

```
6
Order 53 1
Order 67 2
Highest
Serve
Order 72 3
Highest
```

OUTPUT:

```
67
72
```

EXPLANATION:
Orders 53 and 67 are placed with bill amounts 1 and 2 respectively. The highest is order 67 with bill amount 2, which is then served. The order 72 is placed with bill amount 3. There are two orders in waiting 72 and 53 with bill values 3 and 1 and the highest is 72.

Note: No operation other than Order, Highest and Serve is called. Make sure of case sensitive output when printing Invalid

4 Matrix Operations(20 Marks)

Python doesn't have a built-in type for matrices. **However, in this assignment, we will treat a list of a list of floats as a matrix**. Each sub-list inside the outer list is a row in the matrix. In this question you need to implement some matrix operations. We have provided you with a skeleton code that you have to fill, please download that from Gradescope and fill in the functions.You can implement them however you want. The aim of providing it is to make sure you use functions and make them a habit. You have to implement the functions in the exact manner as given in the skeleton code, i.e., you can't change the parameters of a function and their return types. **In this question you don't need to take any input or print any output, we will be calling your functions, you just have to return the values.**

IMPORTANT NOTE: In this question we will **not** be giving any input nor expecting any output. Only the values returned from the functions will be checked so make sure the functions work properly.

An example matrix :- `[[1.00,2.00,3.00],[4.00,5.00,6.00],[7.00,8.00,9.00]]`, the matrix represented by the given list of lists is :-

NOTE: YOU DO NOT WORRY ABOUT FORMATTING OF THE CONTENTS OF THE MATRIX. PERFORM THE OPERATION ON THE MATRIX AS IT IS GIVEN TO THE FUNCTION AND RETURN THE COMPUTED OUTPUT MATRIX WITHOUT ANY FORMATTING FOR DECIMAL PLACES.

$$\begin{bmatrix} 1.00 & 2.00 & 3.00 \\ 4.00 & 5.00 & 6.00 \\ 7.00 & 8.00 & 9.00 \end{bmatrix}$$

4.1 Check Matrix (4 Marks)

This is a function that checks whether the list of list is a matrix or not. A valid matrix has the property that each row has the same number of columns **AND** all the elements of the matrix are float values **AND** every valid matrix has at least one element inside it. Your function will return a Boolean value - True if the list of lists is a matrix and False if it is not. The function name is `CheckMatrix()`. Some samples test cases are -

Example 1:
`Argument`:

```
[[1.00]]
```

`Return`:

```
True
```

Example 2:
`Argument`:

```
[[]]
```

`Return`:

```
False
```

Example 3:
`Argument`:

```
[[1.00,1.00],[2.00]]
```

`Return`:

```
False
```

Example 4:
`Argument`:

```
[[1.00,2.00],[2.00,3.00]]
```

`Return`:

```
True
```

Example 5:
`Argument`:

```
[["Hello",5.00],[]]
```

`Return`:

```
False
```

Example 6:
Argument:

```
[]
```

Return:

```
False
```

Note: This function has to be used in every function after this.

4.2 Transpose (4 Marks)

The function `Transpose()` does transpose of a matrix, it takes a matrix as input and returns its transpose. You also have to check if the list given is a matrix first and **None** must be returned if the list is not a matrix. The list of list returned must have the correct dimensions i.e. if the transpose has dimensions - 5×3 there must be 5 sub lists (or rows) with 3 elements each. Some samples test cases are -

Example 1:
Argument:

```
[[1.00]]
```

Return:

```
[[1.00]]
```

Example 2:
Argument:

```
[[1.00,2.00],[3.00,4.00]]
```

Return:

```
[[1.00,3.00],[2.00,4.00]]
```

Example 3:
Argument:

```
[[1.00,1.00],[2.00]]
```

Return:

```
None
```

Example 4:
Argument:

```
[[1.00,2.00,3.00],[4.00,3.00,9.00]]
```

Return:

```
[[1.00,4.00],[2.00,3.00],[3.00,9.00]]
```

Example 5:
Argument:

```
[["Hello",5.00],[]]
```

Return:

```
None
```

Example 6:
Argument:

```
[[1.00,4.00]]
```

Return:

```
[[1.00],[4.00]]
```

4.3 Addition (4 Marks)

The function `Addition()` takes two lists of lists (matrices) and returns the addition of them (if it exists). If any of the lists of lists is not a matrix or if the operation cannot be done,i.e. the dimnsion of the two matrices are not same, **None** has to be returned. Matrix addition is defined between two matrices if they have the same dimensions.

Matrix addition between two matrices $[a_{ij}]_{m\times n}$ and $[b_{ij}]_{m\times n}$ is defined as $[a_{ij} + b_{ij}]_{m\times n}$

Some samples test cases are -

Example 1:
`Argument:`

```
A = [[1.00]],  B = [[2.00]]
```

`Return:`

```
[[3.00]]
```

Example 2:
`Argument:`

```
A = [[1.00,2.00],[3.00,4.00]], B = [[7.00,8.00],[4.00,1.00]]
```

`Return:`

```
[[8.00,10.00],[7.00,5.00]]
```

Example 3:
`Argument:`

```
A = [[1.00,1.00]], B = [[2.00]]
```

`Return:`

```
None
```

Example 4:
`Argument:`

```
A = [[1.00,2.00,3.00],[4.00,3.00,9.00]], B = [[1.00,4.00],[2.00,3.00],[3.00,9.00]]
```

`Return:`

```
None
```

Example 5:
`Argument:`

```
A = [["Hello",5.00],[]] , B = [[1.00]]
```

`Return:`

```
None
```

Example 6:
`Argument:`

```
A = [[1.00,4.00]],  B = [[0.00,4.00]]
```

`Return:`

```
[[1.00,8.00]]
```

4.4 Multiplication (4 Marks)

The function `Multiplication()` takes two lists of lists (matrices) and returns the multiplication of them (if it exists). If the operation cannot be done **None** has to be returned. Matrix multiplication is defined between two matrices if the dimension of first matrix's column is equal to the dimension of second matrix's row. The multiplication can be done via any algorithm you like. Multiplication won't be performed when either of the list of list is not a matrix. Some samples test cases are -

Example 1:
`Argument:`

```
A = [[1.00]],  B = [[2.00]]
```

`Return:`

```
[[2.00]]
```

Example 2:
`Argument:`

```
A = [[1.00,2.00],[3.00,4.00]],  B = [[7.00,8.00],[4.00,1.00]]
```

`Return:`

```
[[15.00,10.00],[37.00,28.00]]
```

Example 3:
`Argument:`

```
A = [[1.00,1.00]],  B = [[2.00]]
```

`Return:`

```
None
```

Example 4:
`Argument:`

```
A = [[1.00,2.00,3.00],[4.00,3.00,9.00]],  B = [[1.00,4.00],[2.00,3.00],[3.00,9.00]]
```

`Return:`

```
[[14.00,37.00],[37.00,106.00]]
```

Example 5:
`Argument:`

```
A = [["Hello",5.00]],   B = [[]]
```

`Return:`

```
None
```

Example 6:
`Argument:`

```
A = [[1.00,4.00]],   B = [[0.00,4.00]]
```

`Return:`

```
None
```

4.5 Symmetric (4 Marks)

This function `Symmetric()` takes in a list of lists (or matrix) and returns a Boolean value **True** if the matrix is symmetric and **False** if it is not a symmetric matrix. If the list of list given is not a matrix or not a square matrix please return **False**. A symmetric matrix is defined as the square matrix that is equal to its transpose matrix. Symmetric Matrices are only defined for square matrices and return **False** for the rest. Some samples test cases are -

Example 1:
`Argument`:

```
[[1.00]]
```

`Return`:

```
True
```

Example 2:
`Argument`:

```
[[1.00,2.00],[2.00,1.00]]
```

`Return`:

```
True
```

Example 3:
`Argument`:

```
[[1.00,1.00],[1.00]]
```

`Return`:

```
False
```

Example 4:
`Argument`:

```
[[1.00,1.00],[1.00,1.00]]
```

`Return`:

```
True
```

Example 5:
`Argument`:

```
[[1.00,2.00,3.00],[2.00,1.00,7.00]]
```

`Return`:

```
False
```