COMP 1405Z
Fall 2022 – Tutorial #3

Objectives

- Practice writing code using lists
- Practice implementing data structures

Problem 1 – Implementing a Queue

A queue is a data structure used to store and access items using a first-in-first-out ordering. This means that the next item removed from a queue is always the item that has been in the queue the longest (i.e., was added first - think of a line of people waiting to buy something). In this problem, you will use lists to represent queues and write several functions to perform operations on queue lists. Your queue module will need the following components:

1. A global variable representing the maximum size of the queue (i.e., how many items can be stored in the queue at one time). This variable should be assigned a **default value of 10**.
2. An **enqueue(queue, value)** function that accepts a list representing a **queue** and a **value** to be added to the queue**.** The value may be any data type. If there is room in the **queue** list (i.e., the current size is less than the maximum size), **value** will be added to the end of **queue** and the function will return True. If there is no room in the queue, the function will return False.
3. A **dequeue(queue)** function. If the **queue** list has any items in it, this function will remove and return the first item (i.e., from the front of the queue). If **queue** does not have any items in it, the function should return None. In this case, None is the specific Python value representing nothing, not the string value “None”.
4. A **peek(queue)** function. If the **queue** list has any items in it, this function will return the value of the first item in the queue but leave that item in the queue (different from the dequeue function). If **queue** does not have any items in it, the function should return None. In this case, None is the specific Python value representing nothing, not the string value “None”.
5. An **isempty(queue)** function. This function should return True if the **queue** list is empty (no items) and False otherwise.

6. A **multienqueue(queue, items)** function, which takes a **queue** list and a list of elements called **items**. This function should add as many of the elements from the **items** list to the **queue** list as possible (i.e., keep adding until the queue is full or all items have been added) and return the number of items that were added successfully.
7. A **multidequeue(queue, number)** function, which takes a **queue** list and an integer **number**. This function should attempt to dequeue up to **number** items from the queue and return a new list containing all items removed (note: this may be less than **number** if the queue has become empty).

Save your queue function code into a file called **myqueue.py** that **only includes the functions and the maximum size variable** (i.e., no testing code, print statements, etc.). You can test your code by importing your myqueue.py file as a module into a separate Python code file and calling the functions with specific values. To aid in your testing, a queuetester.py file has been included in the tutorial .zip file. If you run this Python file within the same directory as your myqueue.py file, it will perform a number of the queue operations. The testing program will also print out the expected values, so you can verify that your queue functions are working correctly. Note: the queuetester.py file assumes you are treating the start of the queue list as the 'front' of the queue. If you modeled your queue the opposite way (i.e., the end of the list is the 'front' of the queue), the lines that print out the queue will be in reverse order to yours. This is fine as long your queue still works in the first-in-first-out manner required and you still get the same dequeue/queue/peek/etc., results. You should perform additional tests to further verify your functions' correctness.

Problem 2 - Implementing a Stack

In the previous problem, you implemented a queue. A stack is a similar type of data structure, but instead of using the first-in-first-out principle of a queue, a stack uses last-in-first-out. This means that the next item taken from a stack is the most recent item that was inserted. As an example, consider a stack of plates or paper in which you can only look at the top item (peek), add an item to the top of the stack (push), or remove the item from the top of the stack (pop). Create a **mystack.py** file that implements the following functions:

1) **push(stack, value)**: Adds the argument **value** onto the top of the **stack** list.
2) **pop(stack)**: If the **stack** list is empty, this function returns None. Otherwise, this function removes and returns the top value from the **stack** list.
3) **isempty(stack)**: Returns True if the **stack** list is empty (i.e., has no items), False otherwise.
4) **peek(stack)**: If the stack is not empty, returns but does not remove the top value from the **stack** list. Otherwise, returns None.

Save your stack function code into a file called **mystack.py** that **only includes the functions** (i.e., no testing code, print statements, etc.). You can use the **stacktester.py** file from the tutorial zip file to verify the operations of your stack are working as they should be. As with the queue example, the lines that print out the stack contents may print them in the reverse order that you have used, and this is fine so long as your stack is still operating using the last-in-first-out principle. You should test your stack further to ensure it consistently works properly. The next problem will further test your stack implementation.

Problem 3 - Parentheses Validation

A string is considered to have valid parentheses if there are an equal number of opening/closing brackets and each closing bracket – e.g.,), }, or] – matches the most recent, unmatched opening bracket – e.g., (, {, or [. For example:

(({ })) is valid
{ [{ ( ) }] } is valid
(){}[] is valid
((}) is invalid
({) } is invalid

Checking the validity of parentheses is important in verifying the validity of programming code and mathematical statements. Create a new Python file called **validator.py** with a single function called **isvalid(string)**. This function will take a **single string that can consist of any characters** and use your stack implementation from the previous problem to determine if the parentheses are valid or not. The function must return True if the string has valid parentheses and False otherwise. Your function only needs to consider the three types of bracket characters mentioned above – the remaining characters, which could represent code, numbers, arithmetic operators, etc., can be ignored. You can use the validatortester.py file from the tutorial zip file to check whether your function is working correctly. If you are having trouble figuring out how to solve the problem, consider how you would validate each of the sets of brackets above using the stack operations available. An important thing that a stack data structure allows you to do is remember and backtrack through things that you have stored previously (e.g., think of using the back button in a web browser, or undoing operations in a text file). Note that the programming for this problem is not difficult, but the verification algorithm may take some thought.

Submission

Add each of your code files to a single .zip file named "**tutorial3.zip**" and submit it to Brightspace. Ensure that your file is a .zip and has the proper name. Download your submission afterward and check the contents to ensure everything works as expected.