7.1 - Week 7 - Applied Theory

Objectives of this Applied Session

The objectives of this tutorial are to ensure we understand:

- The Stack, Queue and List ADTs
- Big O time complexity of simple algorithms

and we know how to both implement and follow code that combines these concepts.

Recap from Last Week

Based on your understanding of the lessons, answer the following questions in `Ex1.txt`,

1. What is an Abstract Class? Why do we use them?

2. What is an ADT? How is it different from Data types and Data Structures?

An Introduction to Stacks

A *stack* is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top", and the opposite end is known as the "base".

Items that are closer to the base have been in the stack the longest. The most recently added item is always on the top of the stack and thus will be removed first. The stack provides an ordering based on length of time in the collection; the "age" of any given item increases as you move from top to base.

There are many examples of stacks in everyday situations. Consider a stack of plates on a table, where it's only possible to add or remove plates to or from the top. Or imagine a stack of books on a desk. The only book whose cover is visible is the one on top. To access the others, we must first remove the ones sitting on top of them.

Properties Of Stacks

- `Stack()` creates a new, empty stack
- `push(item)` adds the given item to the top of the stack and returns nothing
- `pop()` removes and returns the top item from the stack
- `peek()` returns the top item from the stack but doesn't remove it (the stack isn't modified)
- `is_empty()` returns a boolean representing whether the stack is empty
- `size()` returns the number of items on the stack as an integer

Stack operation	Stack contents	Return value
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4, 'dog']`	
`s.peek()`	`[4, 'dog']`	`'dog'`
`s.push(True)`	`[4, 'dog', True]`	
`s.size()`	`[4, 'dog', True]`	`3`
`s.is_empty()`	`[4, 'dog', True]`	`False`
`s.push(8.4)`	`[4, 'dog', True, 8.4]`	
`s.pop()`	`[4, 'dog', True]`	`8.4`
`s.pop()`	`[4, 'dog']`	`True`
`s.size()`	`[4, 'dog']`	`2`

Application of Stacks

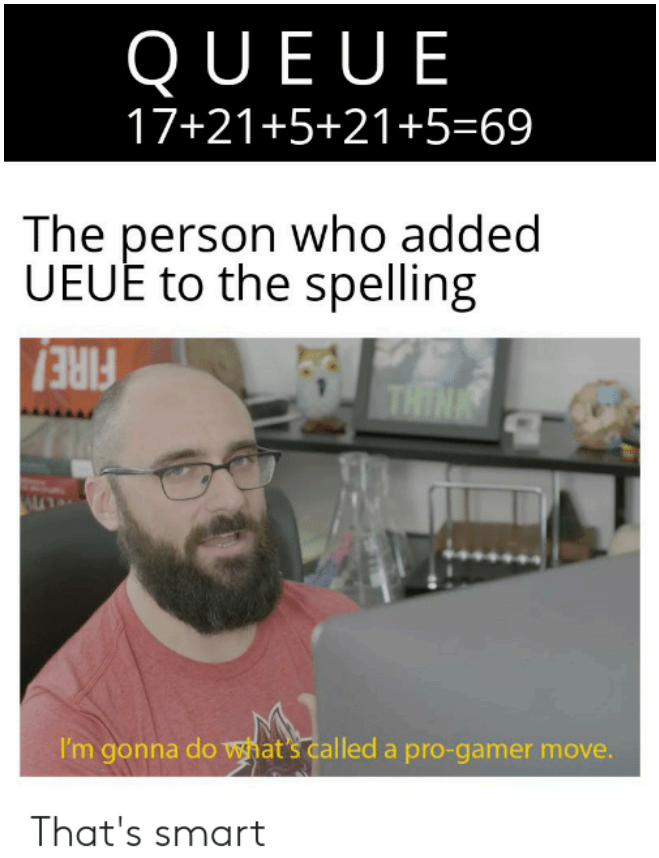
In the lesson on Stack ADTs, we discussed the concept of balanced and unbalanced strings containing parenthesis, square and curly brackets. In `Ex2.py`:

1. Describe (or code if you wish), a Python function to determine if the **parenthesis** ("()") of the input string are balanced. To do that, assume you have a Stack ADT available with the following operations (see `StackADT.py`):
    - Stack(capacity)
    - push(item)
    - pop()
    - is empty()

2. Extend your function to include checks for curly and square brackets.

An Introduction to Queues

What is a Queue?



A queue is an ordered collection of items where the addition of new items happens at one end, called the “rear,” and the removal of existing items occurs at the other end, commonly called the “front.” As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO**, **first-in-first-out**. It is also known as “first-come-first-served.”

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line (so that we can pop the tray stack). Well-behaved lines, or queues, are very restrictive in that they have only one way in and only one way out. There is no jumping in the middle and no leaving before you have waited the necessary amount of time to get to the front.

Queues are a very prevalent model for data flow in real life. Consider an office with 30 computers networked with a single printer. When somebody wants to print, their print tasks “get in line” with all the other printing tasks that are waiting. The first task in is the next to be completed. If you are last in line, you must wait for all the other tasks to print ahead of you.

Operating systems also use a number of different queues to control processes within a computer. The scheduling of what gets done next is typically based on a queuing algorithm that tries to execute programs as quickly as possible and serve as many users as it can. Also, as we type, sometimes keystrokes get ahead of the characters that appear on the screen. This is due to the computer doing other work at that moment. The keystrokes are being placed in a queue-like buffer so that they can eventually be displayed on the screen in the proper order.

Queue Operations

A queue is structured as an ordered collection of items that are added at one end, called the "rear," and removed from the other end, called the "front." The queue operations are:

- `Queue()` creates a new queue that is empty. It needs no parameters and returns an empty queue.
- `append(item)` adds a new item to the rear of the queue. It needs the item and returns nothing.
- `serve()` removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- `is_empty()` tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that `q` is a queue that has been created and is currently empty, then the table below shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item appended so it is the first item returned by serve.

Queue Operation	Queue Contents	Return Value
`q.is_empty()`	`[]`	`True`
`q.append(4)`	`[4]`	
`q.append('dog')`	`['dog', 4]`	
`q.append(True)`	`[True, 'dog', 4]`	
`q.size()`	`[True, 'dog', 4]`	`3`
`q.is_empty()`	`[True, 'dog', 4]`	`False`
`q.append(8.4)`	`[8.4, True, 'dog', 4]`	
`q.serve()`	`[8.4, True, 'dog']`	`4`
`q.serve()`	`[8.4, True]`	`'dog'`
`q.size()`	`[8.4, True]`	`2`

An Application of Queues with Stacks

Consider a **Stack ADT** that implements a stack using some data structure and defines the following methods:

- Stack(capacity)
- pop()
- push(item)
- __len__()
- is_empty()

Consider a **Queue ADT** that implements a queue using some data structure and defines the following methods:

- Queue(capacity)
- serve()
- append(item)
- __len__()
- is_empty()

In `Ex3.py`, describe or code an algorithm **reverse_queue** which would take a queue of strings called **my_queue**, and use stack and queue operations listed above to return a new queue(**result_queue**) containing all non-empty strings from **my_queue** in reverse order, and does this by using a stack. **my_queue** itself must contain the same elements as when it started, and in the same order (i.e., if you cannot think of how to implement the method without modifying **my_queue**, make sure you leave it as it was).

For example, if my_queue has the following 5 elements :
"*Hello*", "*Goodbye*", "*Not now*", "", "*Later*"

where “Hello” is the item at the front, then the method will return the following queue, which has 4 elements with “Later” at the front:

"*Later*", "*Not now*", "*Goodbye*","*Hello*"

A mysterious application?

Consider the following mystery function that uses the StackADT:

```
from my_stack import Stack

def mystery(s):
    sz = len(s)
    s2 = Stack()
    s3 = Stack()

    for _ in range(sz // 2):
        s2.push(s.pop())
    while len(s) > 0:
        s3.push(s.pop())
    while len(s2) > 0:
        s3.push(s2.pop())
    return s3

s = Stack()
s.push(6)
s.push(2)
s.push(7)
s.push(3)

t = mystery(s)
while(len(t) > 0):
    print(t.pop())
```

1. What does the mystery function do? What does this code print?
2. What is the complexity of the mystery function?
3. What would this code return if it used a Queue ADT instead of a Stack ADT? (Replacing pop() with serve() and push() with append())

Complex(ity) Lists

Consider the following code:

```
def mystery (a_list1: List[str], a_list2: List[str]) -> str:
    temp = ""
    n1 = len(a_list1)
    n2 = len(a_list2)
    for i in range(n1):
        if i < n2:
            item2 = a_list2[i]
            temp += item2
            if a_list1[i] > item2:
                return temp
    return temp
```

1. Without executing the code, indicate what the code does and illustrate this with a few different examples.
2. What is the best and worst Big O Complexity of this function for two lists of strings? Explain.

Expanding your Knowledge

This question is about **general ADT design**. Given below is a snippet of a problem description:

> A coffee store in a popular shopping centre is planning to introduce a customer loyalty system. This program is designed to offer rewards for their frequent customers, as well as make their ordering process easier. Each customer has an ID (assigned automatically by the system), name, phone number, and a count of loyalty points. The system will record points earned, accumulated by each customer when they order their coffee. These points can then be redeemed (used) for free coffee on subsequent visits. The system also manages up to 5 different coffees that the store makes. Each coffee has a name and price (in whole dollars). Each coffee is also half price on one day of the week. This day is different for each coffee, and is checked when an order is placed.

In `Ex 6.txt`, describe the ADTs and their associated attributes (no need to program them, just describe them) you would require for a program to implement this customer loyalty system.

As a hint, think about the categories of things that need to be represented in a system, and what operations you need to perform on them. What information does the system track about a customer? How will it manipulate the set of all customers?