جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الانظمة الطبية الذكية

المرحلة الثانية

Lecture: (2)

Subject: Object oriented programming II
Class: Second
Lecturers: Dr. Dunia H. Hameed , Dr. Maytham N. Meqdad

Object Oriented Programming (II) – Second Lecture

1. Overview of Data Structures

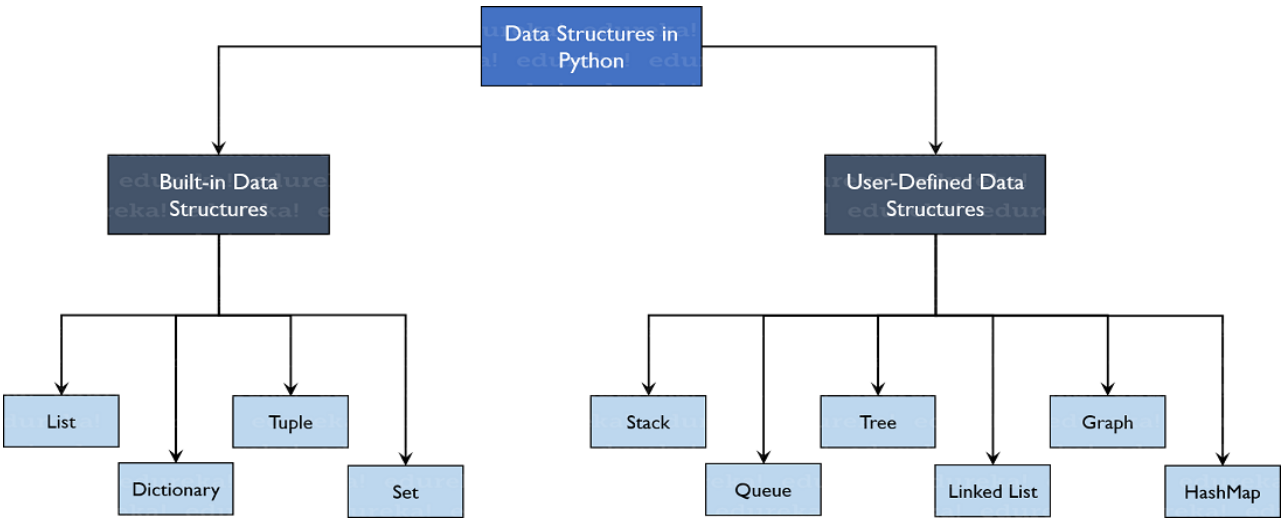
Python has been used worldwide for different fields such as making websites, artificial intelligence and much more. But to make all of this possible, data plays a very important role which means that this data should be stored efficiently and the access to it must be timely. To achieve this, we use something called Data Structures.

Organizing, managing and storing data is important as it enables easier access and efficient modifications. **Data Structures** allows you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.

2. Types of Data Structures in Python

Python has implicit support for Data Structures which enable you to **store and access data**. These structures are called **List, Dictionary, Tuple and Set.**

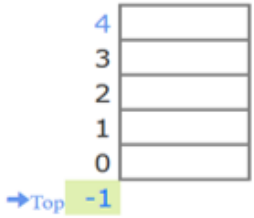
Python allows its users to create their own Data Structures enabling them to have full control over their functionality. The most prominent Data Structures are **Stack, Queue, Tree, Linked List** and so on which are also available to you in other programming languages.



3. User-Defined Data Structures

3.1 Stack

Stacks are linear Data Structures which are based on the principle of Last-In-First-Out (LIFO) where data which is entered last will be the first to get accessed. It is built using the array structure and has operations namely, pushing (adding) elements, popping (deleting) elements and accessing elements only from one point in the stack called as the TOP. This TOP is the pointer to the current position of the stack. Stacks are prominently used in applications such as Recursive Programming, reversing words, undo mechanisms in word editors and so forth.



Problem Description

The program creates a stack and allows the user to perform push and pop operations on it.

Problem Solution

1. Create a class Stack with instance variable items initialized to an empty list.
2. Define methods push, pop and is_empty inside the class Stack.
3. The method push appends data to items.
4. The method pop pops the first element in items.
5. The method is_empty returns True only if items is empty.
6. Create an instance of Stack and present a menu to the user to perform operations on the stack.

Program/Source Code

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, data):
        self.items.append(data)

    def pop(self):
        return self.items.pop()
```

```
s = Stack()
while True:
    print('push <value>')
    print('pop')
    print('quit')
    do = input('What would you like to do? ').split()
```

```
operation = do[0].strip().lower()
if operation == 'push':
    s.push(int(do[1]))
elif operation == 'pop':
    if s.is_empty():
        print('Stack is empty.')
    else:
        print('Popped value: ', s.pop())
elif operation == 'quit':
    break
```

Program Explanation

1. An instance of Stack is created.

2. The user is presented with a menu to perform push and pop operations on the stack.

3. The chosen operation is performed by calling the corresponding method of the stack.

Runtime Test Cases

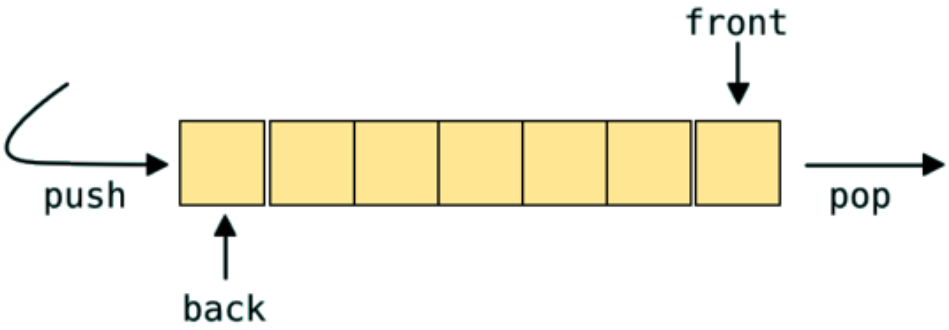
```
Case 1:
push <value>
pop
quit
What would you like to do? push 3
push <value>
pop
quit
What would you like to do? push 5
push <value>
pop
quit
What would you like to do? pop
Popped value:  5
push <value>
pop
quit
What would you like to do? pop
Popped value:  3
push <value>
pop
quit
What would you like to do? pop
Stack is empty.
push <value>
```

```
pop
quit
What would you like to do? quit

Case 2:
push <value>
pop
quit
What would you like to do? pop
Stack is empty.
push <value>
pop
quit
What would you like to do? push 1
push <value>
pop
quit
What would you like to do? pop
Popped value:  1
push <value>
pop
quit
What would you like to do? quit
```

3.2 Queue

A queue is also a linear data structure which is based on the principle of First-In-First-Out (FIFO) where the data entered first will be accessed first. It is built using the array structure and has operations which can be performed from both ends of the Queue, that is, head-tail or front-back. Operations such as adding and deleting elements are called En-Queue and De-Queue and accessing the elements can be performed. Queues are used as Network Buffers for traffic congestion management, used in Operating Systems for Job Scheduling and many more.



Problem Description

The program creates a queue and allows the user to perform enqueue and dequeue operations on it.

Problem Solution

1. Create a class Queue with instance variable items initialized to an empty list.
2. Define methods enqueue, dequeue and is_empty inside the class Queue.
3. The method enqueue appends data to items.
4. The method dequeue dequeues the first element in items.
5. The method is_empty returns True only if items is empty.
6. Create an instance of Queue and present a menu to the user to perform operations on the queue.

Program/Source Code

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, data):
        self.items.append(data)

    def dequeue(self):
        return self.items.pop(0)
```

```
q = Queue()
while True:
    print('enqueue <value>')
    print('dequeue')
    print('quit')
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'enqueue':
        q.enqueue(int(do[1]))
    elif operation == 'dequeue':
        if q.is_empty():
            print('Queue is empty.')
        else:
            print('Dequeued value: ', q.dequeue())
    elif operation == 'quit':
        break
```

Program Explanation

1. An instance of Queue is created.

2. The user is presented with a menu to perform enqueue and dequeue operations on the queue.

3. The chosen operation is performed by calling the corresponding method of the queue.

Runtime Test Cases

```
Case 1:
enqueue <value>
dequeue
quit
What would you like to do? enqueue 3
enqueue <value>
dequeue
quit
What would you like to do? enqueue 1
enqueue <value>
dequeue
quit
```

```
What would you like to do? enqueue 0
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued value:  3
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued value:  1
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued value:  0
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Queue is empty.
enqueue <value>
dequeue
quit
What would you like to do? quit
```

```
Case 2:
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Queue is empty.
enqueue <value>
dequeue
quit
What would you like to do? enqueue 7
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued value:  7
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Queue is empty.
enqueue <value>
dequeue
quit
What would you like to do? quit
```