Quadrant II – Transcript and Related Materials

Programme: B.Sc.
Subject: Computer Science
Semester: III
Paper Code: CSS103
Paper Title: Programming in Python

Unit IV: Object Oriented Concepts
Module Name: Classes, Objects, Abstract Data types
Module No: 13

Name of the Presenter: Mrs. Sonali Sarvesh Karapurkar

OBJECT ORIENTED CONCEPTS

- Python has been an object-oriented language since it existed.
- Because of this, creating and using classes and objects are downright easy.
- This chapter helps you become an expert in using Python's object-oriented programming support.
- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.

- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.
-

CLASS

- A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
- The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Creating Classes

- The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −
- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

class ClassName:

‘Optional class documentation string’

class_suite

EXAMPLE

class Employee:

'Common base class for all employees’

empCount = 0

def __init__(self, name, salary):

self.name = name

self.salary = salary

```
        Employee.empCount += 1

def displayCount(self):

        print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

        print "Name : ", self.name, ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class.
- This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*.
- Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

CREATING INSTANCE OBJECTS

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

EXAMPLE:

```
class Employee:
        'Common base class for all employees’
        empCount = 0
def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
def displayCount(self):
        print "Total Employee %d" % Employee.empCount
def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Output:

Name : Zara ,Salary: 2000

Name : Manni ,Salary: 5000

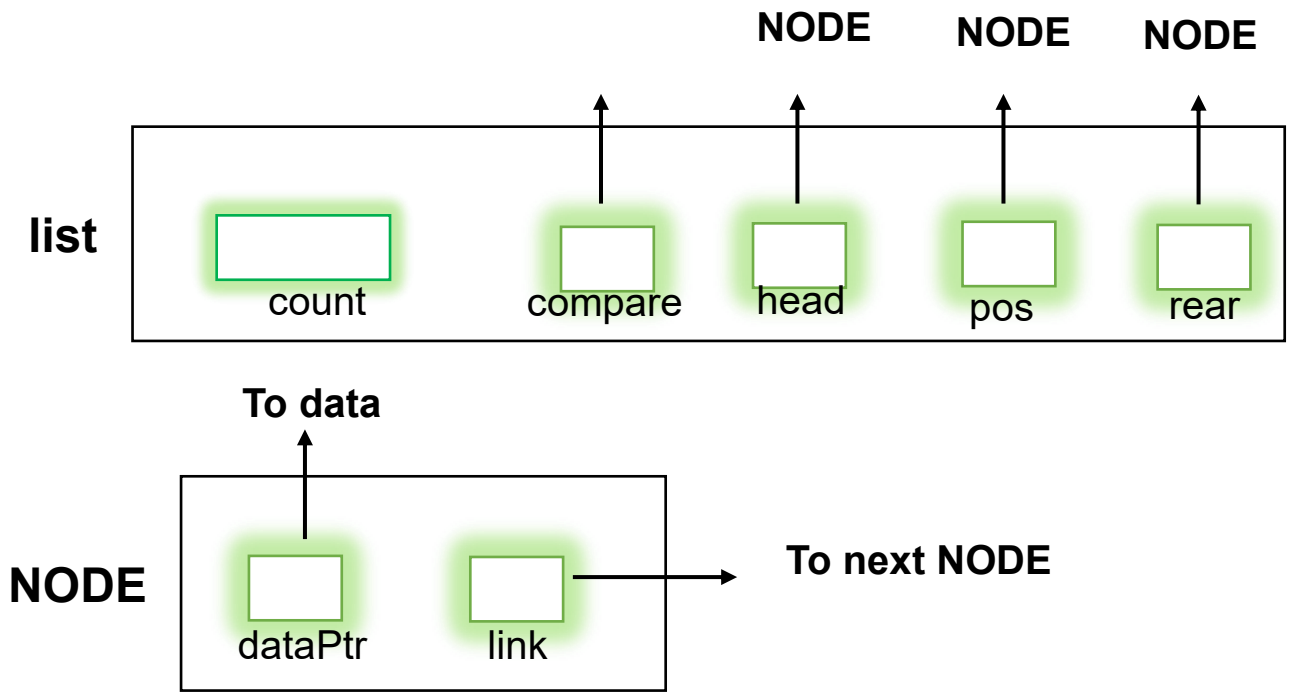
Total Employee 2

ABSTRACT DATA TYPES

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.

1. List ADT

The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.



A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

get() – Return an element from the list at any given position.

insert() – Insert an element at any position of the list.

remove() – Remove the first occurrence of any element from a non-empty list.

removeAt() – Remove the element at a specified location from a non-empty list.

replace() – Replace an element at any position by another element.

size() – Return the number of elements in the list.

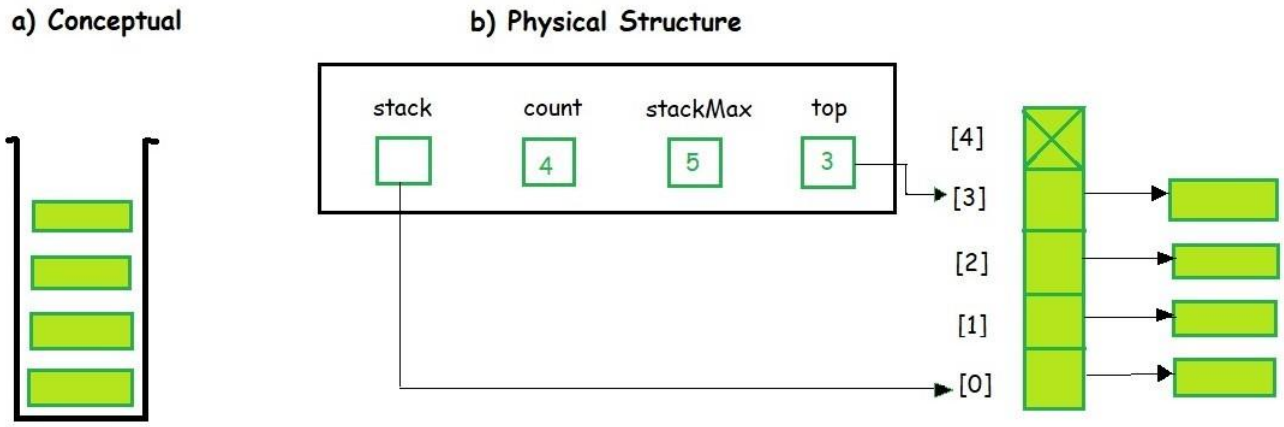
isEmpty() – Return true if the list is empty, otherwise return false.

isFull() – Return true if the list is full, otherwise return false.

2. Stack ADT

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.

The program allocates memory for the *data* and *address* is passed to the stack ADT.



A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

push() – Insert an element at one end of the stack called top.

pop() – Remove and return the element at the top of the stack, if it is not empty.

peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

size() – Return the number of elements in the stack.

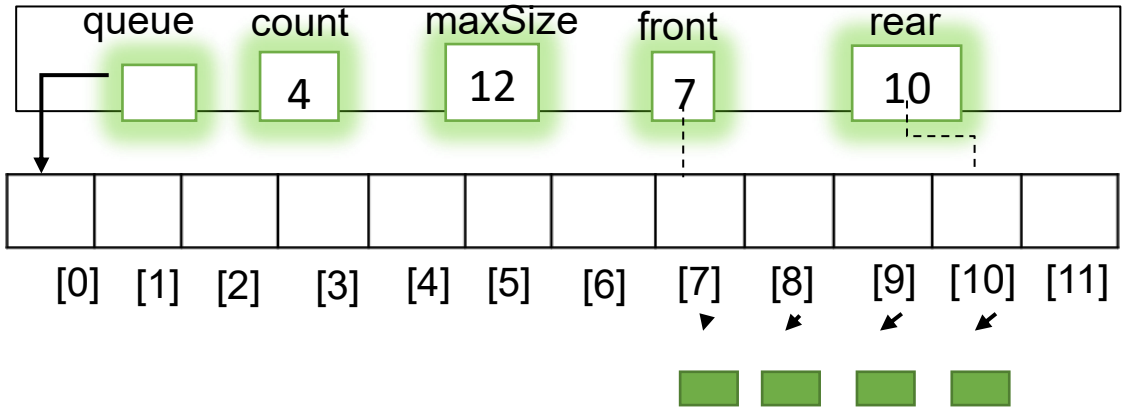
isEmpty() – Return true if the stack is empty, otherwise return false.

isFull() – Return true if the stack is full, otherwise return false.

3. Queue ADT

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.

a) Conceptual



b) Physical Structures

- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue.
- The program's responsibility is to allocate memory for storing the data.

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

enqueue() – Insert an element at the end of the queue.

dequeue() – Remove and return the first element of the queue, if the queue is not empty.

peek() – Return the element of the queue without removing it, if the queue is not empty.

size() – Return the number of elements in the queue.

isEmpty() – Return true if the queue is empty, otherwise return false.

isFull() – Return true if the queue is full, otherwise return false.

Example:

```
from abc import ABC , abstractmethod
class Computer(ABC):
        @abstractmethod
        def process(self):
                pass
class Laptop(Computer):
         def process(self):
                print(“Its Running”)
com1 =Laptop()
com1.process()
```

Output:

Its Running