

# Tutorial - 11

---

TA: Nandika Jain



INDRAPRASTHA INSTITUTE *of*  
INFORMATION TECHNOLOGY **DELHI**



# What is modularity?

---

- Dividing a program into logical parts, with each part doing some well defined computation or providing defined interface and functionality (like use of functions.)
- Cannot write a program of thousands of lines as just one program - must break it into parts and develop parts separately. (**Reusability of code**)
- Easier to debug your code as well.
- Classes and objects is another way to provide modularity, capability to logically divide a problem into computational units. (discussed in the later part of the slides.)

# Data Types

---

- **Basic Data Types:** int, float, bool

On objects of these types, we can do the operations defined on them,

e.g. +, -, \* etc (on int, float), and not/or/and for bool

- **Complex data types:** lists, sets, dictionaries, tuples

We can define variables of this type, and perform the operations on them.

# Data Structures

---

- A data structure is a particular way of organizing data in a computer so that it can be used effectively.

For eg. Storing marks of students, instead of multiple ints, we are storing them in a list.

Stack (LIFO), Linked List, Queue (FIFO)

- Object oriented constructs provide this capability - define new user defined data types.

# Object Oriented Programming

---

- Ability to define a new type: the structure of an object of this type i.e. what attributes/variables it has, and the operations defined on objects of this type.  
(Ability to define variables of this type, and invoke operations on it)
- Can define a "car" as a type (say for a game) which has ops like start, accelerate, turn left, turn right, ... and attributes like color, make, current speed, weight, ...

# Creating a class:

---

- Python allows defining a new type as a class.
- Class is like an object constructor, or a "blueprint" for creating objects.
- **class** myClass:  
    pass

Here, we use the class keyword to define an empty class myClass. From class, we construct instances. An instance is a specific object created from a particular class.

# Object

---

- An object (instance) is an instantiation of a class.
- We can create objects of myClass type and assign them to variables -as with language defined types like dict, sets, lists, int.
- ***p1 = MyClass()***

Role of variable is same as with all types; the nature of object is now of myClass type.

# Methods & Attributes

---

- A class also has methods - functions which define the operations on this new type.
- Only these methods can be executed on objects of this class
- A class has attributes - these are variables in its scope, accessed from within the class to implement the methods of the class.
- Attributes define the state of an object of the class



# Identify Methods and Attributes

---

The data values which we store inside an object are called attributes, and the functions which are associated with the object are called methods.

```
class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age
```

# `__init__`

---

- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- If you don't define it, then: It inherits an `__init__` that takes no arguments other than `self` and does nothing.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created.

# Question!!

---

Create a class named Person, use the `__init__()` function to assign values for name and age:

# Solution:

---

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

# Calling Methods

---

- A class has no state, it is just a definition - the state is within objects.
- When a method is invoked on an object, the function defined in class executes on the state of the object on which it is invoked.
- Operations on an object `obj` are invoked using dot (.) notation:

***object\_variable.operation (args)***

# self

---

- In the method, it is the first parameter self - the ref to the object
- All attributes of an object are accessed as self.attribute
- The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.

# Execution

---

1. When a program is executed, the class definitions are noted by interpreter, but no method is looked at.
2. When an object is created, then the `__init__()` method is visited and executed.
3. When a method is executed on an object, the method function is executed with the ref to the object as the first param.

`<MyClass_obj1>.<method_1>(args)`

The above statement calls the `method_1` defined in `MyClass` with the reference of `MyClass_obj1` in `self`

# Modifying State

---

Set the age of p1 to 40:

```
p1.age = 40
```



# Practice Question 1

---

Design a class called Date. The class should store a date in three integers: month, day, year to initialise.

1) Define getters and setters for this class.

Eg. methods `getmonth()`, `setday(d)` etc.

2) There should be methods to print the date in the following forms:

2/23/2012

February 23, 2012

23 February 2012

3) Create objects of this class and call these methods.

# Solution to Practice Question 1

---

```
import calendar
class Date:
    def __init__(self, month, day, year):
        self.month = month
        self.day = day
        self.year = year
    def get_month(self):
        return self.month
    def get_day(self):
        return self.day
    def get_year(self):
        return self.year
    def set_month(self, month):
        self.month = month
    def set_day(self, day):
        self.day = day
    def set_year(self, year):
        self.year = year
    def print1(self):
        return str(self.month) + "/" + str(self.day) + "/" + str(self.year)
    def print2(self):
        month_name = calendar.month_name[self.month]
        return month_name + " " + str(self.day) + ", " + str(self.year)
    def print3(self):
        month_name = calendar.month_name[self.month]
        return str(self.day) + " " + month_name + " " + str(self.year)
```

#You can use for loop as well instead of the calendar library to compute the month name corresponding to the month number

```
date1 = Date(2,2,2012)
print(date1.print2())
```

#Create more objects and call these methods

# Practice Question 2 (Homework)

---

Define a Movie class that takes Movie Title, Director Name, Genre and IMDB rating to initialize.

1) You may define appropriate getters and setters.

2) Create multiple objects of this class(atleast 5)

3) Run the following queries on these objects:

- Sort Movies by their IMDB rating, in ascending order (You may use the inbuilt functions)
- List Title of Movies with IMDB rating above 7.0
- List Title of Movies directed by a particular director
- List Directors directing a particular genre

Link to sort based on ratings:

<https://stackoverflow.com/questions/3766633/how-to-sort-with-lambda-in-python>

# Solution to Practice Question 2

---

```
class Movie:
    def __init__(self, m_title, d_name, genre, IMDB):
        self.m_title = m_title
        self.d_name = d_name
        self.genre = genre
        self.IMDB = IMDB
    def get_genre(self):
        return self.genre
    def set_genre(self, genre):
        self.genre = genre
    #Create the rest of the getter setter in a similar way
```

```
movie_list = []
movie_list.append(Movie("Inception", "Christopher", "Thriller", 8.2))
movie_list.append(Movie("Gone Girl", "Christopher", "Thriller", 8.5))
movie_list.append(Movie("Annabelle", "John", "Horror", 5.4))
movie_list.append(Movie("Hangover", "Todd", "Comedy", 7.7))
movie_list.append(Movie("Hangover 2", "Todd", "Comedy", 6.5))

#Sorting by IMDB ratings
movie_list.sort(key = lambda x:x.IMDB)
```

---

```
#List movies having IMDB>7
def imdb_7(movie_list):
    new_list = []
    for i in range(len(movie_list)):
        if(movie_list[i].IMDB>7):
            new_list.append(movie_list[i])
    return new_list
```

Similarly, write the other 2 functions.

# Practice Question 3

---

Find the output of the code:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

---

# Stack: LIFO

---

```
class Stack:
    def __init__(self):
        self.top = 0
        self.data = [None]*20
    def push(self, item):
        self.data[self.top] = item
        self.top += 1
    def pop(self):
        if self.top < 1:
            print("Error - popping empty stack")
        item = self.data[self.top-1]
        self.top -= 1
        return(item)
    def isempty(self):
        return self.top <= 0
```

**What will be the changes  
if we have to use the same  
code to implement queue?**

# Queue: FIFO

---

```
class Queue:
    def __init__(self):
        self.qdata = []
        self.front = 0
        self.end = 0

    def add(self, obj):
        self.qdata.append(obj)
        self.end += 1
```

```
    def remove(self):
        if self.isempty():
            return None
        else:
            obj = self.qdata[self.front]
            self.front += 1
            return obj

    def isempty(self):
        if self.front == self.end:
            return True
        else:
            return False
```



# Postfix

---

Let the given expression be “2 3 1 \* + 9 -“. We scan all elements one by one.

- 1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’
- 2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)
- 3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’
- 4) Scan ‘\*’, it’s an operator, pop two operands from stack, apply the \* operator on operands, we get  $3*1$  which results in 3. We push the result ‘3’ to stack. The stack now becomes ‘2 3’.
- 5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get  $3 + 2$  which results in 5. We push the result ‘5’ to stack. The stack now becomes ‘5’.
- 6) Scan ‘9’, it’s a number, we push it to the stack. The stack now becomes ‘5 9’.
- 7) Scan ‘-’, it’s an operator, pop two operands from stack, apply the – operator on operands, we get  $5 - 9$  which results in -4. We push the result ‘-4’ to the stack. The stack now becomes ‘-4’.
- 8) There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).

# Postfix Expression Evaluation using Stack

---

```
def postfix(l):
    estk = Stack()
    for i in l:
        if isinstance(i, float) or isinstance(i, int):
            estk.push(i)
        if isinstance(i, str):
            x = estk.pop()
            y = estk.pop()
            if i == "*":
                estk.push(x*y)
            elif i == "+":
                estk.push(x+y)
            elif ...

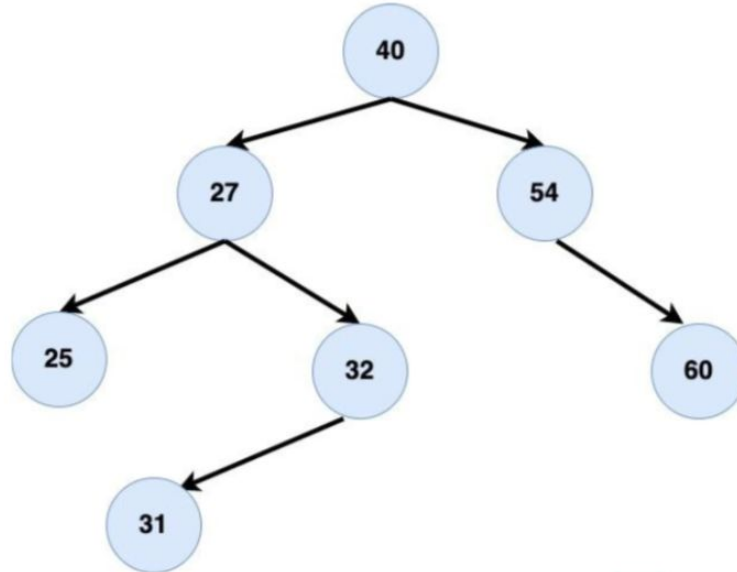
l = [2, 3, 1, "*", "+", 9, "-"]
print(postfix(l))
```

# BST

## Example – Binary Search Tree



- Binary search tree - a root with value, and links to a tree on left and right
- Left sub-tree has values smaller than the root
- Right subtree has values greater than the root
- Searching for an element with binary search tree is very efficient - recursion is natural here
- Printing sorted items is easy



# BST

---

```
class Tree:
    #Attributes: data, ltree, rtree
    def __init__(self, val):
        self.data = val
        self.ltree = None
        self.rtree = None

    def insert(self, val):
        if val == self.data:
            return
        if val < self.data:
            if self.ltree is None:
                self.ltree = Tree(val)
            else:
                self.ltree.insert(val)
        elif val > self.data:
            if self.rtree is None:
                self.rtree = Tree(val)
            else:
                self.rtree.insert(val)
```

```
def ispresent (self, val):
    if val == self.data:
        return True
    if val < self.data:
        print("Search L subtree")
        if self.ltree is None:
            return False
        else:
            return self.ltree.ispresent(val)
    elif val > self.data:
        print("Search R subtree")
        if self.rtree is None:
            return False
        else:
            return self.rtree.ispresent(val)
```

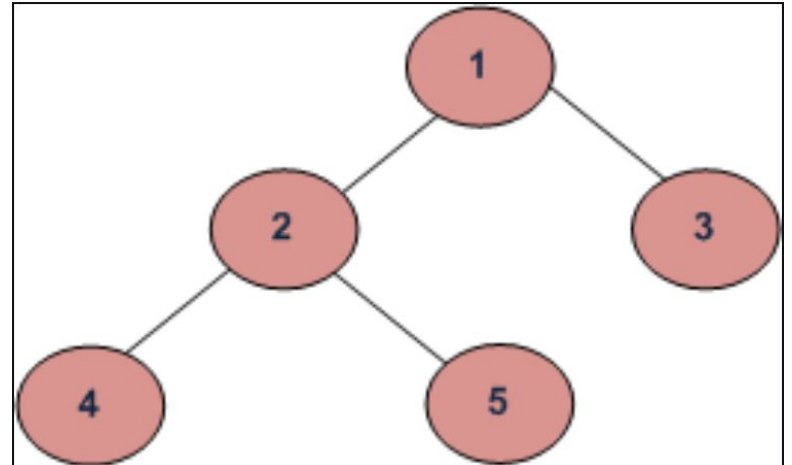
# Tree Traversals

---

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1



# Code for Tree Traversal

---

```
# Print the values in tree in sort
def inorder(self):
    if self.ltree:
        self.ltree.inorder()
    print(self.data)
    if self.rtree:
        self.rtree.inorder()
```

Inorder: Returns in sorted order

**What will be the changes for preorder and postorder??**

# Dunder Methods

---

- In order to provide methods like print and len for use defined classes, we use dunder methods
- Dunder methods- these are methods whose name starts with "\_\_" (double underscore - hence the name)
- Dunder methods cannot be directly called on objects.
- **\_\_str\_\_** :
  - to return a string - containing info about the state of the object. With \_\_str\_\_(), when print(o) is called, \_\_str\_\_() is invoked, which returns a string that gets printed
  - \_\_str\_\_() is also called when str(obj) is called
- **\_\_len\_\_**:
  - Instead of writing a special function for length for a class (e.g. qlen()) , it is better to define this dunder method and use len().

# Dunder Methods

---

More dunder methods:

<https://www.geeksforgeeks.org/customize-your-python-class-with-magic-or-dunder-methods/>

Can we make our own dunder?

<https://stackoverflow.com/questions/9302814/is-it-okay-to-write-own-magic-method>  
s



# Practice Question 4

Find the output of the following code:

`__str__`:

Python objects into strings.

```
mnt > c > Users > Nandika > Desktop > test.py
1  def scale(a,b):
2      a.x = a.x * b
3      print(a.x)
4
5  class myobj:
6      def __init__(self,k=8):
7          self.x = 8
8          print(self.x)
9      def __str__(self):
10         return str(self.x)
11
12
13  s = myobj()
14  t = myobj(4)
15  scale(s,2)
16  print(s)
17  scale(t,2)
18  print(t.x)
```

# Ans to Practice Problem 4

---

```
nandikajain@LAPTOP-L9D5K59L:/mnt/c/Users/Nandika/Desktop$ python3 test.py
8
8
16
16
16
16
```

`__str__` Example:

```
>>> class Car:
...     def __init__(self, color, mileage):
...         self.color = color
...         self.mileage = mileage
...
...     def __str__(self):
...         return 'a {self.color} car'.format(self=self)
...
...
>>> my_car = Car('red', 37281)
>>> print(my_car)
a red car
```

# is and isinstance

---

- **is**: takes two vars and returns True if both vars point to the same object
- **isinstance**(obj, type): returns True if the object obj is of the type type

## ***Example:***

```
emp1 = Employee("John",10000)
```

```
emp2 = emp1
```

```
sal = emp1.incrsalary(10000)
```

```
print(emp1 is emp2)
```

# Object Comparisons

---

- E.g. for `==`, a method `__eq__()` needs to be defined

If you check `==` on objects, without `eq()` definition, python converts it to checking for "is" (i.e. same)

Operation	Dunder method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)

# Queue Example

---

```
# For == operation
def __eq__(self, q2):
    if len(self) != len(q2):
        return False
    for i in range(len(self)):
        if self.qdata[i] != q2.qdata[q2.front+i]:
            return False
    return True
```

```
# for + operation
def __add__(self, q2):
    for i in range(q2.front, q2.end):
        q1.add(q2.qdata[i])
```

# Copying Objects

---

- Class objects are mutable (by defn) - their states can be changed
- So, `obj1 = obj2`, only provides another pointer to `obj`
- For list we have `lst.copy()` method provided by python
- What about copying objects of classes? Can write a `copy()` method
- Better - use the `copy` module provided by python

```
Import copy
```

```
q3 = copy.copy(q1) # copies the queue q1 to q3
```

- `copy()` does a shallow copy - copies only objects, but not nested objects - so they may be pointers
- `copy.deepcopy()` - copies recursively if nested objects

If class definition has a list, what will happen if we use `copy`?

# Class Composition

---

- A class can also use objects of other classes - this allows composition and ability to handle more complex problems
- It enables creating complex types by combining objects of other types
- Objects of a Composite class can contain an object of another class
- In composition, a class definition uses objects of other user defined classes

**Classes define new Types - simple classes use objects of types defined in Python**

Classes can, however, also use objects of other class types - this allows composition, where a composite class is defined using objects of component types

This allows for progressively building higher levels of abstraction / complexity using components

Is a useful method for developing large OO programs

# Class Composition: Example

---

- One class is container and other class is content and if you delete the container object then all of its contents objects are also deleted.

```
class Salary:
    def __init__(self, pay):
        self.pay = pay

    def get_total(self):
        return (self.pay*12)

class Employee:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus
        self.obj_salary = Salary(self.pay)

    def annual_salary(self):
        return "Total: " + str(self.obj_salary.get_total()) + self.bonus

obj_emp = Employee(600, 500)
print(obj_emp.annual_salary())
```



# Practice Question

---

**Problem 4: [Classes]** Define a class Fraction that initialises the object with a list containing the numerator and denominator. Write a program to input two fractions from the user and return comparison of the two fractions.

# Searching

---

## Why searching?

Without them you would have to look at each item of data – each phone number or business address – individually, to see whether it is what you are looking for. In a large set of data, it will take a long time to do this. Instead, a searching algorithm can be used to help find the item of data you are looking for.

## Algorithms:

- **Linear Search**:  $O(n)$

- for i in arr:

- if(i == x):

- return True

# Searching: Binary Search $O(\log n)$

---

- Recursively search in the array.
- Precondition: Array should be sorted.

```
def search(arr, x):  
    low, mid = 0  
    high = len(arr) - 1  
    while low <= high:  
        mid = (high + low) // 2  
        if arr[mid] < x:  
            low = mid + 1  
        elif arr[mid] > x:  
            high = mid - 1  
        else:  
            return mid  
    return -1
```

# Question

---

Convert iterative Binary Search to recursive binary search.

# Answer

---

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}
```

# Sorting

---

## **Why sorting?**

It is easier and faster to locate items in a sorted list than unsorted. Sorting algorithms can be used in a program to sort an array for later searching or writing out to an ordered file or report.

## **Why does Bubble sort work?**

After each iteration, the largest element is pushed towards the end of the array.

# Practice Question

---

Count 1s in a row sorted binary matrix. You're given a  $m \times n$  row sorted binary matrix  $B$ . A row sorted binary matrix is one which can have only 0 or 1 at any index, and where each of the rows is sorted. Write a function **countOnes**, which given  $B$  counts the number of 1s in the matrix.

A trivial solution is to iterate over each element and check whether it is 1 or not. You'll be making  $m \times n$  comparisons using this method. Can you think of something faster using previously discussed concepts?

$L = [[0, 0, 1, 1], [0, 0, 0, 1], [1, 1, 1, 1]]$

`countOnes(L) -> 7`

$L = [[0, 0, 1], [0, 0, 0]]$

`countOnes(L) -> 1`

Can you extend this method to find the row with the maximum number of 1s?

---

```
C:\Users\Nandika>py
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> a = [0, 0 ,1,1, 1]
>>> a.index(1)
2
>>>
```

- Can use count(1) in built method as well.
- Can write a normal iterative loop to find the first index of 1.



# Sorting: Bubble Sort

---

## Algorithm

- Traverse the list till  $n-1$  item
- Compare  $i, i+1$  items - swap the bigger one to right
  - In one traversal biggest item will be the right most
- Then traverse list again but till  $n-2$ ; keep repeating till we reach the start
- This algo is implemented as a nested loop

## #bubble sort

```
def mysort(lst):  
    n = len(lst)  
    for i in range(n-1):  
        for j in range(n-i-1):  
            if lst[j] > lst[j+1]:  
                lst[j], lst[j+1] = lst[j+1],  
lst[j]
```

# Practice Question

---

Bubble sort is stable.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

A number of missiles are coming towards Santa's workshop to destroy all the gifts.

Write a function called **prioritizeMissiles** that accepts an array of Missile objects (represented as dictionaries), and returns a list containing the names of the Missile objects sorted by the amount of time left before the missile hits Santa's workshop. If multiple missiles are going to hit the workshop at the same time, maintain the same order they appear in the passed in array (P.S. look up **stable** sort).

Try to solve this by two ways: (i) Write your own sorting function from scratch.

(ii) The python inbuilt '**sort**' method.

## Some examples

```
Missiles = [{name: 'DASHER_V3', distance: 1000, speed: 100}, {name: 'DANCER_V2.3', distance: 1000, speed: 10}, {name: 'PRANCER_V1.2', distance: 1000, speed: 1}]
```

```
prioritizeMissiles(Missiles) -> ['DASHER_V3', 'DANCER_V2.3', 'PRANCER_V1.2']
```

```
Missiles = [{name: 'VIXEN_V1.1', distance: 500, speed: 4}, {name: 'COMET_V4', distance: 1000, speed: 500}, {name: 'CUPID_V5', distance: 100, speed: 1}]
```

```
prioritizeMissiles(Missiles) -> ['COMET_V4', 'CUPID_V5', 'VIXEN_V1.1']
```