

# OOPs- PYTHON

Dr Sivabalan,  
Technical Training Advisor  
Sivabalan.n@nttdata.com

# Agenda

1. Basics of OOP
2. Types of variables & methods
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Abstraction
7. Interface

What is OOP ?

## Programming Paradigms

- Ways of organizing programs.
- Python supports multiple paradigms. These are as follows:-
  - 1) Procedural oriented paradigm
  - 2) Functional oriented paradigm
  - 3) Object-oriented paradigm

# Requirements

## Object

- An object in OOP represents real-life objects.  
ex. Email, man, student, employee etc
- Every object has two properties.  
1) attributes      2) Behaviours



Attributes:- heading, subject,  
name, recipients list

Behaviours:- sending,  
Adding attachments

## What is class?

- ✓ Class is a template/blueprint/prototype for creating objects.
- ✓ Every object belong to some class
- ✓ Email class:- email1 + email2 + email3 +email4

## What is class?

### attributes:-

heading  
participants  
attachments

### methods:-

send()  
save\_as\_draft()

### ✓ Email1:-

heading:- taking leave  
participant:- xyz  
attachments:- form.pdf

### ✓ Email2:-

heading:- require help  
participant:- abc  
attachments:- pic.jpg



## What is class?

- ✓ Class is a collection of attributes and methods.
- ✓ Class is a collection of objects.
- ✓ Technically, class is a user-defined datatype.



## Creating class and objects

```
class Class_name:  
    #attributes  
    #methods
```

```
obj1 = Class_name([args])  
obj2 = Class_name([args])
```

# OOPs

example.py

```
1 class Employee:
2     def __init__(self,nm,ag):
3         self.name=nm
4         self.age=ag
5     def display(self):
6         print(self.name)
7
8 e1=Employee('raj',21)
9 e2=Employee('jay',22)
```

## What is constructor?

- ✓ Special method used for initializing objects with attributes
- ✓ It is `__init__()` method
- ✓ First arguments is 'self'.

## Types of constructor?

- ✓ Parameterized constructor
- ✓ Non-Parameterized constructor
- ✓ Default Constructor

## How to access class members?

- ✓ Class members :- Attributes(variables) + Actions(Methods)
- ✓ We can access these variables using object outside the class.
- ✓ Syntax:-
  - Accessing attribute:- `object_name.variable_name`
  - Accessing method:- `object_name.method_name()`

## Built-in Class Functions



## Following are built-in class functions:-

- ✓ `getattr(object_name, attribute_name)`
- ✓ `setattr(object_name, attribute_name, new_value)`
- ✓ `delattr(object_name, attribute_name)`
- ✓ `hasattr(object_name, attribute_name)`

## Following are built-in class attributes:-

- ✓ `__dict__` :- Dictionary containing class's namespace
- ✓ `__doc__` :- Class documentation string.
- ✓ `__name__` :- Class Name
- ✓ `__module__` :- Module name in which class is defined
- ✓ `__bases__` :- List of base classes

## Types of variables :-

- ✓ Instance Variables
- ✓ Class Variables

## Instance variables

```
class Student:  
    def __init__( self,nm,m ) :  
        self.name=nm  
        self.marks=m
```

```
std1=Student('Akshay',89)  
std2=Student('Jay',94)  
std3=Student('Ram',91)
```

name=Akshay  
marks=89

name=Jay  
marks=94

name=Ram  
marks=91

## Instance variables:-

- ✓ Variables made for particular instance.
- ✓ Separate copy is created for every object.
- ✓ Values of variables differs from object to object.
- ✓ Modification in one object won't effect other objects.

## Creating instance variables.

- ✓ Using constructor
- ✓ Using instance method
- ✓ Outside class



## Class variables

```
class Student:  
    college='COEP'  
    def __init__( self,nm,m ) :  
        self.name=nm  
        self.marks=m
```

```
std1=Student('Akshay',89)  
std2=Student('Jay',94)  
std3=Student('Ram',91)
```

College='COEP'

name='Akshay'  
marks=89

name='Jay'  
marks=94

name='Ram'  
marks=91

## Class variables:-

- ✓ Variables made for entire class (All objects)
- ✓ Only one copy is created and distributed to all objects
- ✓ Modification in class variable impact on all objects.

## Inheritance in python

## What is inheritance ?

- ✓ Deriving a **new class** from an **existing class** so that **new class inherits all members** (attributes + methods) of existing class is called as inheritance.

- ✓ Old class :- Parent class, Base class, Existing class, Super class

- ✓ New class :- Child class, sub class, derived class



## Creating child class :-

```
class Parent(object):  
    #attributes+methods
```

```
class Child(Parent):  
    #attributes+methods
```



## Bank management system (No Inheritance used):-

**class** Customers:

- SetPersonalDetails()
- GetPersonalDetails()
- SetEducationDetails()
- GetEducationDetails()
- SetBankAccount()

**class** Employee:

- SetPersonalDetails()
- GetPersonalDetails()
- SetEducationDetails()
- GetEducationDetails()
- SetBankAccount()
- SetSalary()
- SetBonus()



## Bank management system (Inheritance used):-

**class** Customers:

- SetPersonalDetails()
- GetPersonalDetails()
- SetEducationDetails()
- GetEducationDetails()
- SetBankAccount()

**class** Employee(Customers):

- SetBankAccount()
- SetSalary()
- SetBonus()

## Constructor in Inheritance

## How constructor works in inheritance

- ✓ By default, constructor of parent class available to child class.

## super() function:-

- ✓ Using super() function, we can access parent class properties.
- ✓ This function returns a temporary object which contains reference to parent class.
- ✓ It makes inheritance more manageable and extensible.

## Types of Inheritance:-

- ✓ Depending on number of child and parent classes involved

- ✓ Single Inheritance
- ✓ Multi-level Inheritance
- ✓ Hierarchical Inheritance
- ✓ Multiple Inheritance
- ✓ Hybrid Inheritance
- ✓ Cyclic Inheritance

## Single Inheritance:-

- ✓ One Parent and one child class





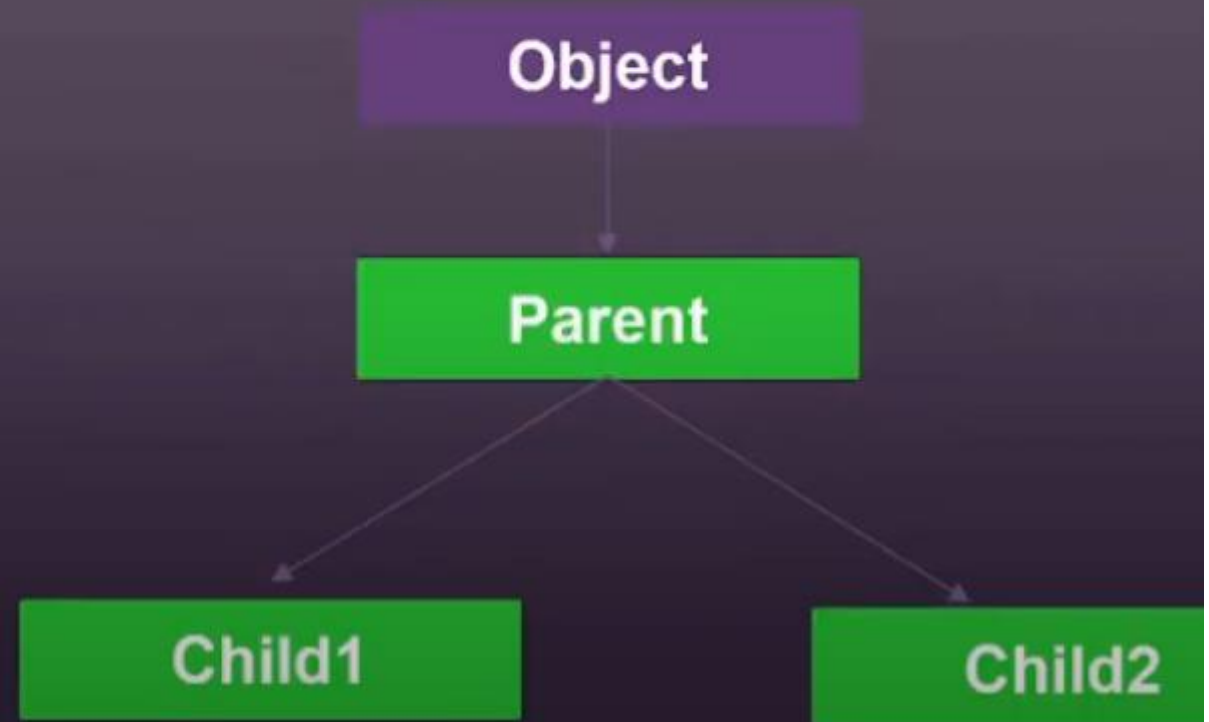
## Multi-level Inheritance:-

- ✓ Parent class and child class further inherited into new class forming multiple levels.



## Hierarchical Inheritance:-

- ✓ One Parent and multiple child classes



## Multiple Inheritance:-

- ✓ Class is derived from multiple base classes.

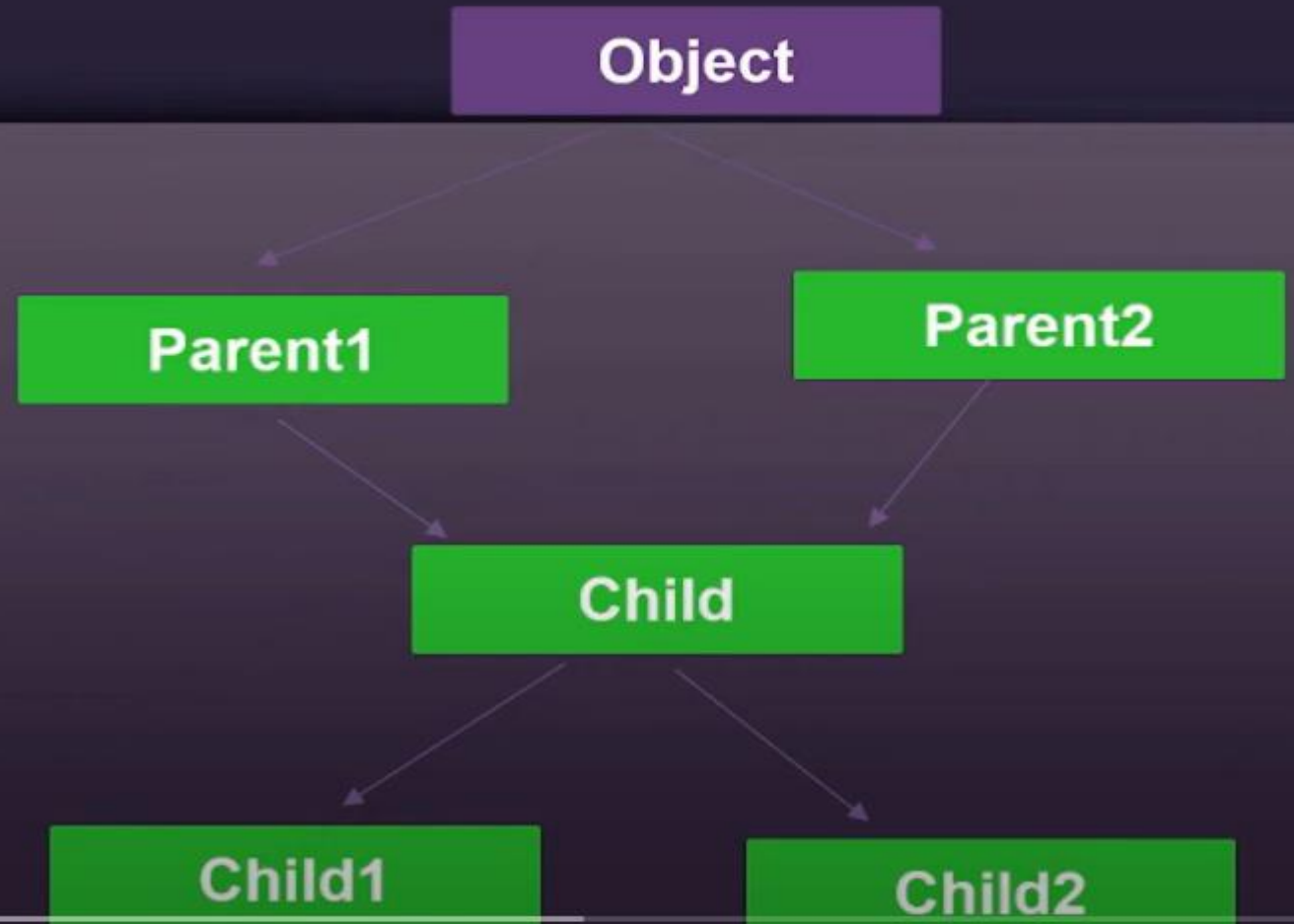


## Syntax:-

```
class Parent1(Object):  
    #parent1 class properties  
class Parent2(Object):  
    #parent2 class properties  
class Child(Parent1,Parent2):  
    #child class properties
```

## Hybrid Inheritance:-

- ✓ It contains multiple type of inheritance.



## What is MRO?

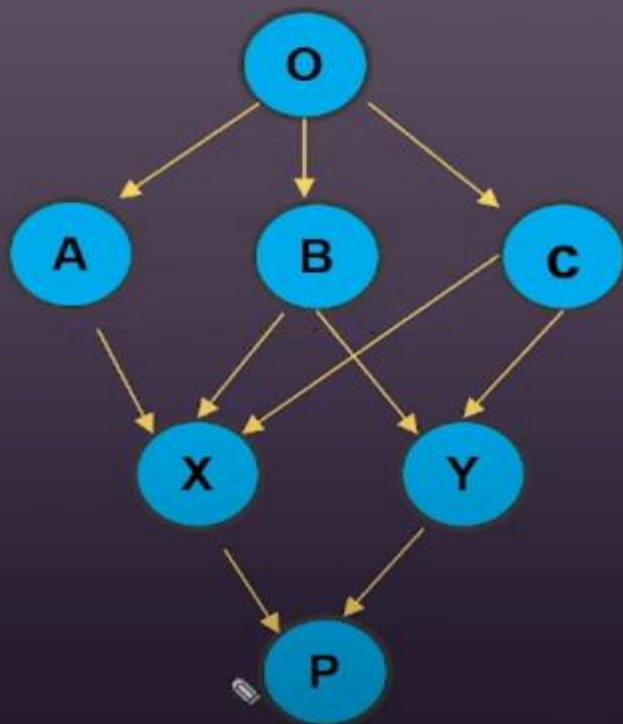
- ✓ MRO represents how properties (attributes+methods) are searched in inheritance.



## Rule -01

- ✓ Python First search in child class and then goes to parent class.
- ✓ Priority is to child class

## Rule -02 MRO Follows 'Depth First Left to Right approach'



- ✓ `mro(o):- Object`
- ✓ `mro(A):- A,O`
- ✓ `mro(B):- B,O`
- ✓ `mro(C):- C,O`
- ✓ `mro(X):- X,A,B,C,O`
- ✓ `mro(Y):- Y,B,C,O`

✓ mro(P):- P,X,Y,A,B,C,O

**Manual Way will  
not give correct  
order always**

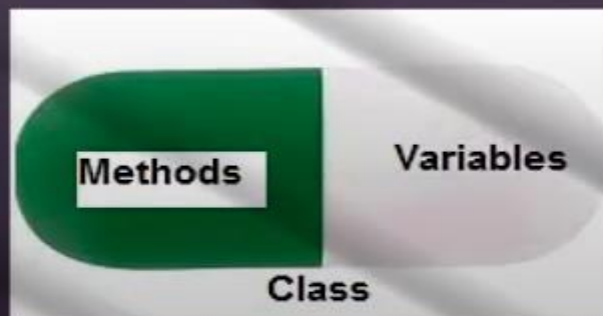
## Encapsulation

## Topics:-

- ✓ What is **Encapsulation** in python?
- ✓ Need of **Encapsulation** in Python
- ✓ **Access Modifiers** in python
- ✓ **Name mangling** concept
- ✓ Making private method

## What is Encapsulation in python?

- ✓ Wrapping up **data and methods working on data together** in a single unit (i.e class) is called as encapsulation.





## Access Modifiers in Python :-

- ✓ Generally, we restrict data access outside the class in encapsulation.
- ✓ Encapsulation can be achieved by declaring the data members and methods of a class as private.
- ✓ Three access specifiers:- public, private, **protected**

## Access Modifiers in Python :-

- ✓ Public member:- Accessible anywhere by using object reference.
- ✓ Private member:- Accessible within the class. Accessible via methods only.
- ✓ Protected member:- Accessible within class and it's subclasses

# Polymorphism

## Topics:-

- ✓ What is Polymorphism in python?
- ✓ Examples of polymorphism
- ✓ Polymorphism in built-in functions

## Real life analogy

You

In front of parent



study, career, exams etc

In front of friends



movies, Netflix, series ,gf-bf etc

## What is Polymorphism in python?

- ✓ Polymorphism in python is an ability of python object to take many forms.
- ✓ If a variable, object, method performs different behaviour according to situation is called as polymorphism.



## Polymorphism with inheritance

## Polymorphism in functions and objects

## Destructor in python

In OOP, we have two terms

➤ Constructor

➤ Destructor



Reverse of each other

## What is Destructor?

- A special method which destroys objects and releases resources tied to objects.
- Destructor is called automatically when object is destroyed.

# What is purpose of Destructor ?

- Releasing objects tied to destroyed objects

X = 100

Y = 200

# database connection

# cache created

# file handling done



X

Y

DB object  
Cache vars  
Handler  
etc



Below are two conditions when destructor is called :-

- Reference counting reaches to 0.
- When variable goes out of scope

Note:- In Python, The special method `__del__()` is used to define a destructor.

## Object Creation & Deletion in python:-

```
emp = Employee('Jay',50000)
```



Arguments passed here for  
constructor to initialize variables

Object is created using  
**\_\_new\_\_()** method

Object is initialized using  
**\_\_init\_\_()** method

```
del emp
```

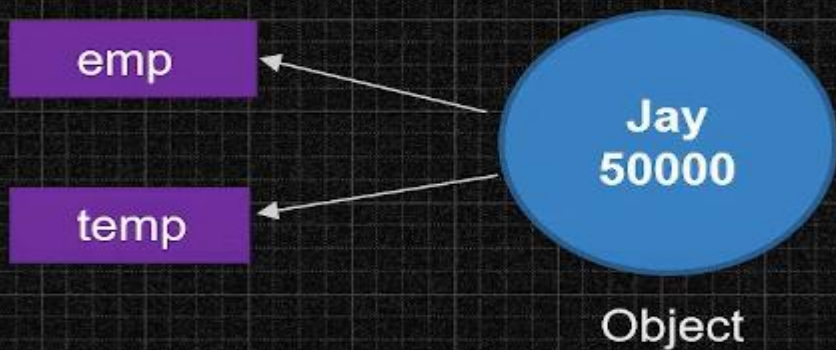
Destructor invoked using  
**\_\_del\_\_()** method



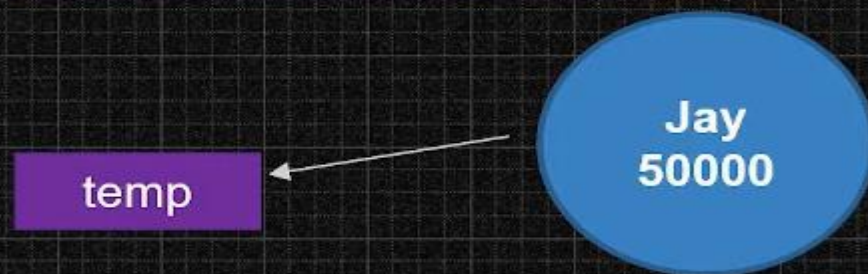
## Working of Destructor in python:-

### 1. Create Object

```
emp = Employee('Jay',50000)  
temp=emp
```



### 2. delete emp :- del emp



`__del__()` method will not be invoked as we still have one reference

### 3. delete temp :- del temp

Object is ready for garbage collection, call the destructor