Object-Oriented Programming in Python →

In Python object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphism encapsulation etc. in the programming. The main concept of oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.

OOPS Concepts in Python →

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python Class →

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Creating a class and object with class and instance attributes

```
Class Employee:
        name = "Punit"  # Class attributes
        def __init__(self, name):
                Self.name = name

Emp 1 = Employee ("Rahul")
Emp2 = Employee ("Sahil")
print("My name is {}".format (Emp1.name))
print("My name is {}".format (Emp2.name))
```

Output ->

```
        My name is Rahul
        My name is Sahil
```

Creating classes and objects with methods ->

```
Class Employee:
        attr 1 = "Employee"
        def __init__(self, name):
            self.name = name
        def speak (self):
                print("My name is {}".format(self.name))

Emp 1 = Employee (" Nishant")
Emp2 = Employee(" Sourabh")
Emp 1. speak()
Emp 2. speak()
```

Output ->

```
        My name is Nishant
        My name is Sourabh
```

Some Points on Python Class →

- Classes are created by keyword class.
- Attributes are the variables that belong to a class. Attributes are always public and can be accessed using the dot(.) operator.

eg → My Class. My Attribute

Python objects →

The object is an entity that has a state and behaviour associated with it. It may be any real world objects like a table, pen etc. Integers, strings, floating point numbers, even arrays, and dictionaries all are objects.

An object Consists of →

- State → It is represented by the attributes of an object. It also reflects the properties of an object/class.
- Behaviour → It is represented by the methods of an object. It also reflects the response of an object to other.
- Identity → It gives a unique name to an object and enables one object to interact with other objects.

Inheritance Syntax ->

```
Class BaseClass_Name():
        # statement 1
        |
        |
        # statement N

Class DerivedClass_Name(BaseClass_Name):
        # statement 1
        |
        |
        # statement N
```

Example ->

```
# Parent Class
Class Person(object):
    def __init__(self, name, id):
        self.name = name
        self.id = id

    def display(self):
        print("My name is {}".format(self.name))
        print("Id number: {}".format(self.id))
```

The python --init-- method →

The --init-- method is similar to constructors in C++ and Java. It is called as soon as an object of an class is instantiated. The method is useful to do any initialization you want to do with your object.

Python Inheritance →

In python, Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.

Features of inheritance are →

- It represent real world relationships well.
- It provide the reusability of a code. We don't have to write the same code again and again.
- It allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

```
# Child Class
 Class Employee (Person):
     def __init__(self, name, id, salary, post):
            self.salary = salary
            self.post = post
     Person.__init__(self, name, id)

     def details (self):
            print ("My name is {}".format(self.name))
            print(" Id no: {}".format(self.id))
            print(" Salary: {}".format(self.salary))
            print(" Post : {}". format(self.post))

 Emp1 = Employee ("Mohit", 120, 30000, "Intern")
 Emp1.details()
```

Output →

```
My name is Mohit
Id no : 120
Salary : 30000
Post : Intern
```

Types of Inheritance →

1) Single Inheritance →
Single level inheritance enables a derived class to inherit characteristics from a single parent class.

2) Multilevel Inheritance →
Multilevel inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherit properties from his parent class.

3) Hierarchical Inheritance →
Hierarchical Inheritance enables more than one derived class to inherit properties from a single parent class.

4) Multiple Inheritance →
Multiple-level Inheritance enables one derived class to inherit properties from more than one base class.

Polymorphism example →

```
Class India():
        def capital(self):
            print("New Delhi is the capital of India")
        def type(self):
            print("India is a developing country")

Class USA():
        def capital(self):
            print("Washington D.C. is the concept of USA")
        def type(self):
            print("USA is a developed country")

obj-ind = India()
obj-usa = USA()
for country in (obj-ind, obj-usa):
        country.capital()
        country.type()
```

Output →

```
New Delhi is the capital of India.
India is a developing country.

Washington D.C is the capital of USA.
USA is a developed country.
```

Python Polymorphism →

In python, polymorphism simply means having many forms. This code demonstrates the concept of Python oops inheritance and method overriding in Python classes. It shows how subclasses can override methods defined in their parent class to provide specific behaviour while still inheriting other methods from the parent class.

Syntax →

```
Class BaseClass-Name:
    def fun1():
        # statements

Class Child1-Name:
    def fun1 ():
        # statements

class Child2-Name:
    def fun2 ():
        # statements.
```

Encapsulation Example ->

```
Class Base:
    def --init--(self):
        # protected member
        self._a = 2

Class Derived(Base):
    def --init--(self):
        Base.--init--(self)
        print("Calling protected member of Base Class")
        print(self.--a)
    # modify the protected variable
        self.--a = 3
        print("Calling modified protected member
                 outside the class:", self.--a)


obj1 = Base()
obj2 = Derived()
print("Accessing protected member of obj1:", obj1.--a)
print("Accessing protected member of obj2:", obj2.--a)
```

Output ->

```
Calling protected member of base class: 2
Calling modified protected member outside class: 3
Accessing protected member of obj 1: 2
Accessing protected member of obj 2: 3
```

Python Encapsulation →

In python, encapsulation is one of the fundamental concepts in OOPs. It describe the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an objects method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables etc.

Syntax →

```
Class ClassName:
    # Statement 1
        :
        :
        :
    # Statement N

Object_Name = ClassName()
```

Data Abstraction example →

```
from abc import ABC, abstractmethod
Class car (ABC):
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    @abstractmethod
    def printDetails(self):   #abstract Method
        pass
    def accelerate(self):   #concrete Method
        print("Speed up....")
    def break_applied(self):
        print("car stopped")

Class Hatchback(car):
    def printDetails(self):
        print("Brand:", self.brand)
        print("Model:", self.model)
        print("Year:", self.year)

    def sunroof(self):
        print("Not having sunroof")
```

Python Data Abstraction →

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive part of our code implementation and this is where data abstraction come.

Data Abstraction can be achieved by creating abstract classes in Python, Abstract classes can be created using abc (abstract base class) module and abstract method of abc module.

Abstract Class →

Abstract class is a class in which one or more abstract methods are defined. When a method is declared inside the class without its implementation is known as abstract method.

Create abstract Base class and Abstract Method →

1) Firstly, we import ABC and abstract method class from abc (Abstract Base class) library.
2) Create a Base Class that inherits from The ABC class. In python when a class inherits from ABC, it indicates that the class is intended to be an abstract base class.

```
Car 1 = Hatchback("Maruti", "Alto", "2022")
Car 1.print Details()
Car 1.accelerate()
Car 1.sunroof()
```

Output →

```
Brand : Maruti
Model : Alto
Year : 2022
Speed up...
Not having sunroof
```

3) Inside Base class we declare an abstract method named "method-1" by using "abstract method" decorates. Any subclass derived from Base Class must implement this method-1 method. we write pass in this method which indicates that there is no code or logic in this method.

Syntax →

```
from abc import ABC,
abstractmethod
class Baseclass(ABC):
    @ abstractmethod
    def method-1 (self):
        # empty body
        pass
```

Concrete Method →

Concrete methods are the methods define in an abstract base class with their complete implementation. Concrete methods are required to avoid reprication of code in subclass.