@learneverythingai

INTRODUCING PYTHON

OPPS CONCEPTS

SHIVAM MODI
@learneverythingai

CLASSES AND OBJECTS

In Python, classes are like blueprints for creating objects. Objects are instances of classes, with their own unique data and characteristics. Let's see some code:

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

    def bark(self):
        return "Woof!"

# Creating an instance of the Dog class
my_dog = Dog("Buddy", 3)
print(my_dog.name)  # Output: Buddy
print(my_dog.bark())  # Output: Woof!
```

SHIVAM MODI
@learneverythingai

ENCAPSULATION

Encapsulation means hiding the internal details of an object and providing a controlled interface. Use access modifiers like public, private, and protected to control data visibility.

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

# Creating a BankAccount instance
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

SHIVAM MODI
@learneverythingai

INHERITANCE

Inheritance allows you to create a new class based on an existing class. The new class inherits attributes and methods from the base class. It's great for code reusability!

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

my_dog = Dog("Buddy")
print(my_dog.make_sound())  # Output: Woof!
```

SHIVAM MODI
@learneverythingai

POLYMORPHISM

Polymorphism allows different classes to be treated as instances of the same class through a shared interface. It's all about flexibility and adaptability!

```
def animal_sound(animal):
    return animal.make_sound()

my_cat = Cat("Whiskers")
print(animal_sound(my_dog))  # Output: Woof!
print(animal_sound(my_cat))  # Output: Meow!
```

SHIVAM MODI
@learneverythingai

CONSTRUCTORS AND DESTRUCTORS

Constructors (init) are special methods that are automatically called when an object is created. Destructors (del) are called when an object is about to be destroyed.

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        print(f"A {brand} {model} is created.")

    def __del__(self):
        print(f"The {self.brand} {self.model} is being destroyed.")

my_car = Car("Toyota", "Corolla")
del my_car  # Output: The Toyota Corolla is being destroyed.
```

SHIVAM MODI
@learneverythingai

METHOD OVERRIDING

Method overriding allows a subclass to provide a specific implementation of a method that's already defined in its superclass. This adds flexibility to the codebase.

```python
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

my_circle = Circle(5)
my_rectangle = Rectangle(4, 6)
print(my_circle.area())  # Output: 78.5
print(my_rectangle.area())  # Output: 24
```

SHIVAM MODI
@learneverythingai

ABSTRACT BASE CLASSES

Abstract Base Classes provide a blueprint for other classes to follow. They help enforce certain methods and structures, ensuring consistency across derived classes.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

my_square = Square(7)
print(my_square.area())  # Output: 49
```

SHIVAM MODI
@learneverythingai

COMPOSITION

Composition involves creating objects that are composed of other objects. It's a powerful way to build complex systems and maintain modular code.

```python
class Engine:
    def start(self):
        return "Engine started."

class Car:
    def __init__(self):
        self.engine = Engine()

    def start_engine(self):
        return self.engine.start()

my_car = Car()
print(my_car.start_engine())  # Output: Engine started.
```

SHIVAM MODI
@learneverythingai

@learneverythingai

SHIVAM MODI
@learneverythingai

LIKE THIS POST?

- *Follow Me*
- *Share with your friends*
- *Check out my previous posts*

www.learneverythingai.com

Follow

SHARE

SHIVAM MODI
@learneverythingai