OOPS

(I accidentally joined PES)

in Python

Hitesh Pranav

- OOPs
- Instance variables / Objects
- Class variables
- Scope of variables
- hasattr, getattr, setattr, delattr
- static & class method
- Inheritance

What is OOPs ?

- The programming paradigm based on the concept of objects.
- These objects bundle data & behaviour into a single unit
  - data → attributes / properties
  - behaviour → methods / functions
    - In OOPs, methods word is used instead of functions
- OOPs have 4 Key aspects
  - **Abstraction** :
    Focus on essential characteristics of object & hide the irrelevant ones
  - **Encapsulation** :
    Combines data & behaviour within object, protecting from external manipulation
  - **Inheritance**
    Allows creating new classes based on existing ones, inheriting their attributes & methods
  - **Polymorphism**
    Enables objects of different classes respond to same msg in different ways

Defining a new class

```python
class Car:
    color = "red"
    speed = 0

    def accelerate(self, amount):
        self.speed += amount
        print(f"Car is now going {self.speed} km/h!")

    def brake(self):
        self.speed -= 5
        print("Car is slowing down...")

my_car = Car()

my_car.accelerate(20)
my_car.brake()
```

Class variables → They are shared by all the objects in the class

Instance variables → They are unique to each object

```
Car is now going 20 km/h!
Car is slowing down...
```

Mentioning that the object is from the class

Mentioning class name

Scope of variables

- 2 Types :
  - Local Variable
    - Defined inside a function & exist within that function's execution
  - Global Variable
    - Defined outside a function & accessible throughout that function's execution

Note : Instance Variable > Class Variable
↳ higher priority

__init__ method

- It is a method (also called 'constructor') which automatically gets called when a new object is being created

```
class Car:
    def __init__(self,color, engine, fuel_level =100):
        self.color = color
        self.speed = 0
        self.engine = engine
        self.fuel_level = fuel_level

    def accelerate(self,amount):
        self.speed += amount
        print()
    def brake(self):
        self.speed -=5
        print("Car is slowing down...")


my_car = Car('blue','v8')
print(my_car.color)
print(my_car.speed)
print(my_car.engine)
print(my_car.fuel_level)
```

```
blue
0
v8
100
```

Self is a keyword which is used to reference the current object itself. It helps differentiate object's attributes, methods and external variables & functions

Mentioning the diff variables we want to take input of

Defining each input we got into a variable

Printing out output of the object

Understanding hasattr, getattr, setattr, delattr

- **hasattr(object, attribute)**
  - Checks if object has a specific attribute or not

```
class Car:
    '''def __init__(self,color, engine, fuel_level =100):
        self.color = color
        self.speed = 0
        self.engine = engine
        self.fuel_level = fuel_level'''

    def accelerate(self,amount):
        self.speed += amount
        print()
    def brake(self):
        self.speed -=5
        print("Car is slowing down...")


my_car = Car()
if hasattr(my_car, 'color'):
    print(f'My car is {my_car.color}')
else:
    print('Car has no color attributed')
```

→ has been commented

```
Car has no color attributed
```

→ Used hasattr

- **getattr (object, attribute, default = None)**
  - Retrieves value of an object

```
class Car:
    def __init__(self,color, engine, fuel_level =100):
        self.color = color
        self.speed = 0
        self.engine = engine
        self.fuel_level = fuel_level

    def accelerate(self,amount):
        self.speed += amount
        print()
    def brake(self):
        self.speed -=5
        print("Car is slowing down...")


my_car = Car('blue','v8')

my_car.accelerate(25)
my_car.brake()

current_speed = getattr(my_car,'speed',0)
print(f"My car's speed is {current_speed} km/h")
```

```
Car is slowing down...
My car's speed is 20 km/h
```

→ getattr

→ If no value, default is taken as mentioned

- setattr(object, attribute, value)
  - Sets the value of an object's attribute

```
class Car:
    def __init__(self,color, engine, fuel_level =100):
        self.color = color
        self.speed = 0
        self.engine = engine
        self.fuel_level = fuel_level

    def accelerate(self,amount):
        self.speed += amount
        print()
    def brake(self):
        self.speed -=5
        print("Car is slowing down...")


my_car = Car('blue','v8')

setattr(my_car,'speed',45)
print(f"My car is going at {getattr(my_car,'speed')} km/h")
```

```
My car is going at 45 km/h
```

setattr used

- delattr(object, attribute)
  - Deletes an attribute from object

```
class Car:
    def __init__(self,color, engine, fuel_level =100):
        self.color = color
        self.speed = 0
        self.engine = engine
        self.fuel_level = fuel_level

    def accelerate(self,amount):
        self.speed += amount
        print()
    def brake(self):
        self.speed -=5
        print("Car is slowing down...")


my_car = Car('blue','v8')

setattr(my_car,'speed',45)
delattr(my_car,'speed')
print("My car's speed attribute has been deleted")
```

```
My car's speed attribute has been deleted
```

setattr & delattr

@classmethod (AIN'T IN SYLLABUS)

- It performs operations on class itself & not on specific instances
- Can be called directly on class without need for object creation
- Typically used for:
  - Creating objects with different initializations (Alternative Constructor)
  - Creating objects based on different criteria or configurations
  - Performing tasks related to class but not any instance (Utility Functions) (Factory Method)

```python
class Car:
    def __init__(self, color, engine, transmission, fuel_level):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.fuel_level = fuel_level


    @classmethod
    def create_from_data(cls,color, engine, transmission, fuel_level):
        return cls(color,engine,transmission,fuel_level)


my_car = Car.create_from_data('blue','v8','automatic',90)
print(f"My car is a {my_car.color} sports car with a {my_car.engine} engine having {my_car.fuel_level}% fuel!")
```

Using classmethod for creating an alternative constructor

```
My car is a blue sports car with a v8 engine having 90% fuel!
```

```python
class Car:
    def __init__(self, color, engine, transmission, mileage):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.mileage = mileage


    @classmethod
    def create_for_city_driving(cls):
        return cls("silver","v6","Automatic", 25)


my_car = Car.create_for_city_driving()
print(f"My car is {my_car.color} with {my_car.transmission} transmission!")
```

Using classmethod for Factory method

```
My car is silver with Automatic transmission!
```

@staticmethods

- It is similar to regular functions, but bound to the class which can be called directly on the class or an object instance

- Typically used for:
  - Providing general functionalities which are not directly related to class or instances → Utility Functions
  - Calculations & Data Manipulation helping used by classes & instances
  - Keeping related functions grouped within class for code organization

```python
class Car:
    def __init__(self, color, engine, transmission, mileage):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.mileage = mileage

    @staticmethod
    def calculate_distance(speed,time):
        return speed*time

    @staticmethod
    def license_plate_validation(plate):
        state,no1,letters1,no2=plate.split(' ')
        if state == 'KA' or 'AP' or 'MH' or 'KL' or 'TN':
            if int(no1) > 0 and int(no1)< 99:
                if int(no2) > 0 and int(no2) < 9999:
                    return True
        else:
            return False

my_trip_distance = Car.calculate_distance(60,5)
print(f'I travelled {my_trip_distance} kms')

is_valid_true = Car.license_plate_validation('KA 14 L 23')
print(f'License plate valid? {is_valid_true}')
```

```
I travelled 300 kms
License plate valid? True
```

Both @staticmethod are bound to the class automatically

- Difference b/w @classmethod & @staticmethod
  - Class methods require name of class
    Static methods doesn't
  - Class methods operate on class itself
    Static methods are more general & used for any purpose
  - Class methods are associated with class but can create objects
    Static methods are independent of objects & simply bound to class

Inheritance

- Concept in OOPs that allows to create new classes based on existing classes
- Its like building a foundation on an existing class and inheriting its properties & functionalities and adding own unique features to create more specialized version.

```python
class Car():
    def __init__(self,color,engine,fuel_level):
        self.color =  color
        self.engine = engine
        self.fuel_level = fuel_level
    def start_engine(self):
        print('Vehicle is starting')

class Vehicle(Car):
    def __init__(self,color,wheels,doors,engine,fuel_level):
        super().__init__(color,engine,fuel_level)
        self.doors = doors
        self.wheels = wheels

    def horn(self):
        print('Beep Beep!')

    def accelerate(self):
        print(f'The {self.engine} car is accelerating')

my_car = Vehicle('blue',4,4,'V8',100)
my_car.start_engine()
my_car.horn()
my_car.accelerate()
```

```
Vehicle is starting
Beep Beep!
The V8 car is accelerating
```

→ Parent Class

→ Child Class which inherited from parent class

→ A built-in function that helps access methods & properties of parents class even when they are overridden in child class

Exception handling

- What is exception?
  - An exception is an event which disrupts normal flow of program
- Why Handle exceptions ?
  - Improve Program Stability
  - Better user experience
  - Easier debugging
- 3 Exception Handling Constructs
  - try
    - Block contains the code which needs to be protected from errors
  - except
    - It allows to define how to respond to an error in the code
  - else
    - Runs only if no exception occurs in try block
  - finally
    - Runs regardless of whether exception occurs or not

```python
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Invalid input! Please enter a number.")
else:
    print(f"You entered: {age}")
finally:
    print('age is just a number .-.')
```

maincode

when error

When no error

```
Enter your age: 18
You entered: 18
age is just a number .-.
```

```
Enter your age: hi
Invalid input! Please enter a number.
age is just a number .-.
```