Python Fundamentals (Part4)

Concepts : Object Oriented Programming in Python

Object Oriented Programming

Let's suppose we need to build a software to store all the information related to students studying in our college.

These students could have a lot of **properties** (like name, parent's name, address, graduation year, cgpa etc.) associated with them.

They could also have a lot of **behaviours** (like fee payment, scholarship calculation, checking minimum attendance criteria etc.) associated with them.

Without existing Python data types, we can choose to store this information in lists or dictionaries but that would not be efficient. As we have a lot of properties, we'd have to create too many lists and it'll be hard to keep track of them. If we use dictionaries, we'd have to re-define the same keys again & again & we won't logically be able to associate behaviours with that data.

So to simplify our work of creating such a software we have **Object Oriented Programming** - which gives us the concept of **classes** & **objects**.

What is a Object Oriented Programming (OOP)?

Object-oriented programming (OOP) in Python is a programming paradigm centered around organizing code into **objects:** bundles of data (attributes) and behaviour (methods). It helps us structure programs in a way that is modular, reusable, and easier to maintain.

Note - It's not compulsory to always use OOP concepts but they definitely help in a lot of cases.

OOP models software as a collection of interacting **objects**, similar to how we think about real-world entities. Each object represents something meaningful in our program - such as a *user*, a *car*, a *bank account*, or an *enemy* in a video game.

In Python, OOP is based around **classes**, which are blueprints for creating objects.

What are Classes & Objects?

Class

A class is a blueprint or template for creating objects. We can think of a class as a recipe: it describes what an object will look like (its attributes) and what it can do (its methods), but it is not the object itself.

rijumandal2810@gmail.com

```python
class Car:
    brand = "Toyota"
```

Object

An **object** (or **instance**) is a realization of a class, it is the actual thing created based on the class blueprint. Now based on the template we can create as many objects as we want.

```python
car1 = Car()
car2 = Car()

print(car1.brand)      # Toyota
print(car2.brand)      # Toyota
```

We use the "." (Dot Operator) to access properties & methods of objects.

Class vs. Object

Class	Object
Class is a blueprint/ template.	An object is a concrete instance of a class.
Does not exist in memory until instantiated.	Contains actual data & occupies memory.
One class can create any number of objects.	Each object is independent.

Attributes & Methods

Attributes are variables and **Methods** are functions defined inside a class.
(We'll cover both in detail)

Constructor in OOP

In Python, a **constructor** is a special method used to initialize newly created objects. It is not responsible for creating the object, instead the constructor sets up the object with initial values.

We use the `__init__(self, ...)` method to define our constructor. Whenever we create an object of a class, Python automatically calls the `__init__()` method.

```python
class Student:
    def __init__(self):
        print("constructor was called")

stu1 = Student()         # "constructor was called"
```

rijumandal2810@gmail.com

`self` is a special parameter which refers to the **instance of the class** that is calling the method. We don't need to pass it explicitly.

We can also use constructor to initialize values for objects:

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

stu1 = Student("Rahul")
stu2 = Student("Harshita")

print(stu1.name, stu2.name)   # Rahul Harshita
```

Types of Constructors

1. **Default Constructors** - A constructor with **no parameters** except `self`.
2. **Parameterized Constructors** - Takes parameters to initialize values uniquely for each object.

Note - Python doesn't support constructor overloading directly (like Java/C++) i.e. having multiple constructors in the same class. Whichever is written last is executed.

Attributes in OOP

Attributes are **variables** that belong to a class or an object. They store data/state of the object.

Types of Attributes

1. **Class Attributes**

- Belong to the **class itself**, shared by *all* objects.
- Defined **outside** any method in the class.

```
class Student:
    college = "ABC college"          # class attribute

stu1 = Student ()

print(stu1.college)
print(Student.college) # class attribute can also be accessed with class name
```

rijumandal2810@gmail.com

2. Instance Attributes

- Belong individually to **each object**.
- Defined inside the `__init__` method using `self` .
- Each object gets its own copy.

```
class Student:
    def _init_(self, name, gpa): # instance attributes
        self.name = name
        self.gpa = gpa

stu1 = Student("Rahul", 8.7)
print(stu1. name, stu1.gpa)
```

Methods in OOP

Methods are **functions defined inside a class**, representing the **behaviour** or **actions** of an object.

Types of Methods

1. Instance Methods

- Take `self` as the first argument.
- Can access both **instance attributes** and **class attributes**.

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

    def display(self):          # Instance method
        print(f"Name: {self.name}, Marks: {self.marks}")
```

2. Class Methods

- Use `@classmethod` decorator.
- Take `cls` (class) as first argument.
- Used to work with **class-level data**.

```
class Student:
    school_name = "ABC School"

    @classmethod
    def change_school(cls, new_name):
        cls.school_name = new_name
```

rijumandal2810@gmail.com

3. Static Methods

- Use `@staticmethod` decorator.
- Do **not** take `self` or `cls` .
- Behave like normal functions but belong to the class for logical grouping.

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b
```

OOP Pillars

Let's understand the 4 Key pillars of OOP - Encapsulation, Abstraction, Inheritance & Polymorphism.

Encapsulation

Encapsulation is the bundling of data (variables) and methods (functions) that operate on that data into a single unit (a class), along with controlling access to that data. This is done to protect the data from accidental or unauthorized modification.

To implement encapsulation, we use access modifiers. Python has 3 access levels:

1. Public members

- Accessible everywhere, written like normal variables.

```
class Student:
    def __init__(self, name):
        self.name = name  # public variable

s = Student("Rahul")
print(s.name)  # Allowed
```

2. Protected members

- Indicated by a **single underscore** `_` (suggest - "Don't access directly unless needed.")
- Still accessible from outside (not truly protected).
- Intended for internal use or inheritance.

rijumandal2810@gmail.com

```python
class Person:
    def __init__(self):
        self._age = 20  # protected variable

p = Person()
print(p._age)  # Technically allowed, but not recommended
```

3. Private members

- Indicated by a **double underscore** `__`
- Python does *name mangling*: the variable name becomes `_ClassName__variable`.
- Cannot be accessed directly from outside.

```python
class Bank:
    def __init__(self, balance):
        self.__balance = balance # private variable

b = Bank(5000)
print(b.__balance)  # ERROR: attribute not accessible
```

To access:

```python
print(b._Bank__balance) # Allowed (name-mangled form)
```

Python uses naming conventions with access modifiers, not strict enforcement (like Java/C++).

Getters & Setters

When we make a variable private, we use methods to read it (getters) or update it (setters).

```python
class Employee:
    def __init__(self, salary):
        self.__salary = salary       # private

def get_salary(self):                # getter
    return self.__salary

def set_salary(self, new_salary):    # setter
    self.__salary = new_salary

e = Employee(50000)
print(e.get_salary())
e.set_salary(60000)
```

rijumandal2810@gmail.com

Inheritance

Inheritance is where one class (child) acquires the properties and behaviors (variables + methods) of another class (parent).

The class whose properties are inherited - **Parent** / Base / Superclass

The class that inherits - **Child** / Derived / Subclass

```
class Employee:                # parent class
    start_time = "9AM"
    end_time = "5PM"

class Teacher(Employee):       # child class
    def __init__(self, subject):
        self.subject = subject

t1 = Teacher("Data Science")

print(t1.subject, t1.start_time, t1.end_time)
```

Inheritance enables:

- Code reuse
- Extensibility
- Cleaner, maintainable design
- Polymorphism

Types of Inheritance

1. Single Inheritance

A child inherits from one parent.

```
# Parent → Child

class Parent:
    def display(self):
        print("Parent class")

class Child(Parent):
    pass

c = Child()
c.display() # Output: Parent class
```

rijumandal2810@gmail.com

2. Multi-level Inheritance

A child inherits from a parent, and another class inherits from the child.

```
# Grandparent(Employee) → Parent(AdminStaff) → Child(Accountant)

class Employee:
    start_time = "9AM"
    end_time = "5PM"

class AdminStaff(Employee):
    def __init__(self, role):
        self.role = role

class Accountant(AdminStaff):
    def __init__(self, salary, role):
        super().__init__(role)
        self.salary = salary

acc1 = Accountant(50_000, "CA")

print(acc1.salary, acc1.role, acc1.start_time, acc1.end_time)
```

3. Multiple Inheritance

A child inherits from more than one parent class.

```
class Teacher:
    def __init__(self, salary):
        self.salary = salary

class Student():
    def __init__(self, gpa):
        self.gpa = gpa

class TA(Teacher, Student):
    def __init__(self, name, salary, gpa):
        super().__init__(salary)            # call parent constructor
        Student.__init__(self, gpa)         # call parent constructor
        self.name = name

ta = TA("Rahul", 50_000, 7.5)

print(ta.name, ta.salary, ta.gpa)
```

super() keyword - Used to call parent class's method from child class.

Abstraction

Abstraction is hiding unnecessary implementation details and showing only the essential features to the user.

rijumandal2810@gmail.com

Example - In real life when we drive a car & press the breaks, the car stops. But we don't need to know how the hydraulic systems work. To implement same idea in Python, we have abstraction.

We implement abstraction with abstract classes & abstract methods.

Abstract Class

An abstract class in Python is one which:

- Cannot be instantiated
- Can contain normal + abstract methods
- Usually acts as a blueprint for child classes

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
```

Abstract Method

It is a method declared but not implemented (Children **must** override abstract methods).

```
@abstractmethod
def method_name(self):
```

Example of Abstraction

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound():
        pass

class Lion(Animal):
    def make_sound(self):
        print("Roar!")

class Cow(Animal):
    def make_sound(self):
        print("Moo!")

lion = Lion()
lion.make_sound()

cow = Cow()
cow.make_sound()
```

rijumandal2810@gmail.com

Polymorphism

Polymorphism is the ability of a single function, operator, or object to behave differently based on the context. (“poly” = many, “morph” = forms)

The idea is that:

- Same method name - works differently for different objects
- Same operator - behaves differently depending on operand types

Let’s look at an example:

```
print(1 + 2)    # adds 2 numbers
print("1" + "2") # concatenates 2 strings
```

Same ‘+’ operator being used for 2 different operations, called **Operator Overloading.**

Let’s look at two popular types of polymorphism:

1. **Function Overriding** (or Method Overriding)

- When a child class provides its own version of a method that already exists in the parent class (Both methods should have same name)
- Type of Runtime Polymorphism (dynamic binding)
- Child method **takes precedence** over parent method.

```
class Animal:
    def sound(self):
        print("Some generic sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

a = Animal()
dog = Dog()

a.sound() # Some generic sound
dog.sound() # Bark
```

rijumandal2810@gmail.com

2. Duck Typing

- Works on the idea: *"If it looks like a duck and quacks like a duck, it must be a duck."*

```python
class Dog:
    def speak(self):
        print("Bark")

class Cat:
    def speak(self):
        print("Meow")

class Robot:
    def speak(self):
        print("Beep Boop")

def make_it_speak(entity):
    entity.speak()  # doesn't care about type

d = Dog()
c = Cat()
r = Robot()

for e in [d, c, r]:
make_it_speak(e)
```

> *Keep Learning & Keep Exploring!*

rijumandal2810@gmail.com