Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a method of writing programs using **objects** and **classes**. It helps make code more organized, reusable, and easier to understand.

Why use OOP?

- Organize code into **real-world entities**
- Make code **modular and reusable**
- Improve **data security and structure**
- Reduce code duplication and improve maintainability

Key Concepts in Python OOP

1. **Class**: A blueprint to create objects. It defines properties (variables) and methods (functions).
2. **Object**: An instance of a class. It represents a real-world entity.
3. **Encapsulation**: Hiding internal details and only exposing what is necessary.
4. **Abstraction**: Showing only essential features and hiding complex details.
5. **Inheritance**: One class can use properties and methods of another class.
6. **Polymorphism**: Same function name behaves differently for different classes.

Simple Example:

```
class Car:
    name = "Neeraj"
    game = "Card Game"

    def start(self):
        print("Car started")

my_car = Car()
print(my_car.name)
my_car.start()
```

In the example above:

- `Car` is a **class**
- `my_car` is an **object**
- `start()` is a **method**

What is a Constructor

- It is a function that runs **automatically** when an object is created.
- In Python, the constructor method is named `__init__()`.
- It is used to assign **initial values** to object properties (like `name`, `age`, etc.).

TECHVISION-TECHNOLOGY

Syntax of Constructor in Python

```
class ClassName:
    def __init__(self, parameters):
        # code to initialize the object
```

Example of Constructor

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

    def show(self):
        print("Name:", self.name)
        print("Marks:", self.marks)

s1 = Student("Neeraj", 90)    # object created, constructor
runs
s1.show()
```

Output:

```
Name: Neeraj
Marks: 90
```

Types of Constructors in Python:

Type	**Description**
Default Constructor	Constructor with no parameters
Parameterized Constructor	Constructor with parameters to initialize values

Default Constructor Example:

```
class Demo:
    def __init__(self):        # No parameters
        print("Object Created")

obj = Demo()
```

Parameterized Constructor Example:

```
class Demo:
    def __init__(self, message):    # With parameter
        print("Message:", message)

obj = Demo("Hello Python")
```

Summary:

- `__init__()` is a **constructor**
- It runs **automatically** when an object is created
- Used to **initialize** object variables

TECHVISION TECHNOLOGY

What is a Decorator

A **decorator** in Python is a function that **adds extra functionality** to another function **without changing its structure**.

Think of it like "wrapping" a function inside another function to **extend its behavior**.

Why Use Decorators:

- Code **reuse**
- Add **logging**, **authentication**, **timing**, etc.
- Cleaner and more Pythonic way to modify functions

Basic Structure of a Decorator:

```
def decorator_function(original_function):
    def wrapper_function():
        print("Before the function runs")
        original_function()
        print("After the function runs")
    return wrapper_function
```

Example Without @ Syntax:

```
def greet():
    print("Hello!")

def decorator(func):
    def wrapper():
        print("Welcome")
        func()
        print("Goodbye")
    return wrapper

decorated_greet = decorator(greet)
decorated_greet()
```

Output:

```
Welcome
Hello!
Goodbye
```

Example With @decorator Syntax:

```
def decorator(func):
    def wrapper():
        print("Welcome")
        func()
        print("Goodbye")
    return wrapper

@decorator
def greet():
```

TECHVISION-TECHNOLOGY

```
    print("Hello!")

greet()
```

Decorator with Arguments:

```
def decorator(func):
    def wrapper(name):
        print("Welcome")
        func(name)
        print("Goodbye")
    return wrapper

@decorator
def greet(name):
    print("Hello", name)

greet("Neeraj")
```

Example Using a Function with Arguments:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Arguments passed:", args, kwargs)
        result = func(*args, **kwargs)
        print("Function executed successfully.")
        return result
    return wrapper

@my_decorator
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Neeraj")
greet("Anjali", greeting="Hi")
```

Output:

```
Arguments passed: ('Neeraj',) {}
Hello, Neeraj!
Function executed successfully.

Arguments passed: ('Anjali',) {'greeting': 'Hi'}
Hi, Anjali!
Function executed successfully.
```

Summary:

Term	**Meaning**
`@decorator`	Shortcut to apply a wrapper to a function

TECHVISION-TECHNOLOGY

Term	Meaning
`wrapper()`	Inner function that modifies the behavior
Useful For	Logging, timing, access control, etc.

Getters and Setters in Python

Getters are methods used to **get/access** the value of a private attribute.
Setters are methods used to **set/update** the value of a private attribute.
They help in **encapsulation**, which is one of the core concepts of **Object-Oriented Programming (OOP)**.

Why Use Getters and Setters?

- To **protect** direct access to private attributes.
- To **validate** data before setting it.
- To **control** how an attribute is accessed or modified.
- To implement **read-only** or **write-only** properties.

Syntax (Using Methods):

```
class Student:

    def __init__(self):
        self.__name = ""

    # Getter method
    def get_name(self):
        return self.__name

    # Setter method
    def set_name(self, name):
        if len(name) > 0:
            self.__name = name
        else:
            print("Name cannot be empty")

# Usage
s = Student()
s.set_name("Alice")
print(s.get_name())
```

Using `@property` Decorator (Pythonic Way)

```
class Student:
    def __init__(self):
        self.__name = ""
```

TECHVISION-TECHNOLOGY

```
    @property
    def name(self):            # Getter
        return self.__name

    @name.setter
    def name(self, value):     # Setter
        if len(value) > 0:
            self.__name = value
        else:
            print("Invalid name")
# Usage
s = Student()
s.name = "Bob"        # Calls setter
print(s.name)         # Calls getter
```

Key Points:

Feature	**Explanation**
`__var`	Double underscore makes variable private
`@property`	Turns method into a getter
`@varname.setter`	Defines setter for the same property
Encapsulation	Prevents direct access to attributes
Validation	Can be added inside setter

Inheritance in python:

Code **reusability**: You don't have to write the same code again.
It helps in **organizing** code in a hierarchical manner.
You can **extend** or **modify** behavior of the parent class in the child class.

Basic Syntax of Inheritance:

```
class Parent:
    def display(self):
        print("This is Parent class.")

class Child(Parent):   # Inheriting Parent class
    def show(self):
        print("This is Child class.")

obj = Child()
```

TECHVISION-TECHNOLOGY

```
obj.display()   # Accessing Parent class method
obj.show()      # Accessing Child class method
```

Types of Inheritance in Python:

Type	Description	Example
Single	One child class inherits from one parent class	`class B(A)`
Multiple	One child inherits from more than one parent class	`class C(A, B)`
Multilevel	One class inherits from a child class which in turn inherits from another	`class C(B), class B(A)`
Hierarchical	Multiple child classes inherit from the same parent class	`class B(A), class C(A)`
Hybrid	Combination of more than one type of inheritance	Combination of above

Access Modifiers in Python:

In Python, **access modifiers** are used to **control the visibility (accessibility)** of class members (variables and methods). Unlike some other languages like Java or C++, Python doesn't have strict access control, but it provides **naming conventions** to simulate them.

Types of Access Modifiers in Python:

Modifier	Syntax Example	Access Level
Public	`self.name`	Accessible from anywhere
Protected	`self._name`	Suggests access within class and subclass
Private	`self.__name`	Not accessible outside the class (name mangled)

TECHVISION TECHNOLOGY

1. Public Members:

Can be accessed **anywhere** (inside or outside the class).

```
class Student:
  def __init__(self):
    self.name = "Neeraj" # Public member

s = Student()
print(s.name) # Accessible
```

2. Protected Members:

Defined with a **single underscore _**.
By convention, should not be accessed outside the class or its subclasses.
Python doesn't enforce this restriction.

```
class Student:
  def __init__(self):
    self._marks = 90 # Protected

class Derived(Student):
  def show(self):
    print("Marks:", self._marks)

d = Derived()
d.show()
print(d._marks) # Technically accessible, but not recommended
```

3. Private Members:

Defined with **double underscore __**.
Python uses **name mangling** to make them harder to access from outside.

```
class Student:
  def __init__(self):
    self.__roll = 101 # Private

  def show(self):
    print("Roll No:", self.__roll)

s = Student()
s.show()
# print(s.__roll)  # Error: 'Student' object has no attribute
'__roll'
print(s._Student__roll) # Accessible via name mangling (not
recommended)
```

Summary Table:

Modifier	Prefix	Access From Outside	Subclass Access	Notes
Public	none	Yes	Yes	Default access
Protected	_var	Yes (by convention no)	Yes	Just a naming convention
Private	__var	No (name mangled)	No (directly)	Still accessible via workaround

Tip:

Python focuses more on **developer discipline** than strict enforcement. Use private/protected when needed, but don't rely on them for security.

1. Normal Methods (Instance Methods)

TECHVISION-TECHNOLOGY

These are the most common type of methods in Python classes.

- Take `self` as the **first parameter**.
- Can access **instance variables and methods**.
- Can access **class variables** using `self` or `ClassName`.

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def show(self):  # normal method
        print("Value is:", self.value)

obj = MyClass(10)
obj.show()
```

2. Static Methods:

These methods are not tied to a class instance

- **Do not take `self` or `cls`** as the first argument.
- **Cannot access instance or class variables** directly.
- Use `@staticmethod` decorator to define.

When to use:

Use static methods when you want a function that belongs to a class **logically**, but **doesn't need** to access or modify class/instance data.

Example:

```
class Math:
    @staticmethod
    def add(x, y):  # static method
        return x + y

print(Math.add(5, 3))  # no need to create object
```

TECHVISION-TECHNOLOGY

Instance Variables vs Class Variables:

1. Instance Variables:

- Defined **inside the constructor** (`__init__`) using `self`.
- **Unique for each object**.
- Belong to **instances** (objects), **not** the class itself.

Example:

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

s1 = Student("Neeraj", 90)
s2 = Student("Aman", 85)

print(s1.name) # Neeraj
print(s2.name) # Aman
```

2. Class Variables

- Defined **outside** all methods but **inside** the class.
- **Shared by all objects** of the class.
- Belong to the **class**, not to individual objects.
- You can access them using either ClassName.var or self.var (but modifying via self creates a new instance variable).

Example:

```
class Student:
  school = "TechVision" # class variable

  def __init__(self, name):
    self.name = name      # instance variable

s1 = Student("Neeraj")
s2 = Student("Aman")

print(s1.school) # TechVision
print(s2.school) # TechVision

Student.school = "New School" # changing class variable
print(s1.school) # New School
```

Summary Table:

Feature	**Instance Variable**	**Class Variable**
Defined in	Constructor (`__init__`)	Inside class, outside methods
Belongs to	Object (instance)	Class (shared by all instances)
Accessed via	`self.variable`	`ClassName.variable` or `self.variable`
Storage	Unique copy per object	Single copy shared by class
Used for	Object-specific data	Common data for all objects

TECHVISION-TECHNOLOGY

What is a Class METHOD?

- A class method is a method that works with the class, not the instance.
- It takes cls as the first argument instead of self.
- It is used to access or modify class variables.

Example:

```
class Student:
    school_name = "Tech Vision Technology"

    def __init__(self, name):
        self.name = name

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

    def show(self):
        print(f"Name: {self.name}, School:
{Student.school_name}")


# Before changing school
s1 = Student("Neeraj")
s1.show()  # Tech Vision Technology

# Change class variable using class method
Student.change_school("Future Tech School")
s1.show()  # Future Tech School
```

When to use `@classmethod` ?

- You want to access or modify class variables.
- You are writing factory methods that return instances of the class.

What Is an Alternative Constructor?

An **alternative constructor** is a class method that returns an instance of the class using **custom logic** (like parsing a string, reading from a file, or processing data) instead of calling `ClassName(...)` with normal arguments.

Example: Using @classmethod as an Alternative Constructor:

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

    @classmethod
```

TECHVISION-TECHNOLOGY

```
    def from_string(cls, data_str):
        name, age = data_str.split('-')
        return cls(name, int(age))

# Creating object normally
p1 = Person("Neeraj", 23)

# Creating object using alternative constructor
p2 = Person.from_string("Neeraj-23")

print(p1.name, p1.age)  # Neeraj 23
print(p2.name, p2.age)  # Neeraj 23
```

Why Use Class Methods as Alternative Constructors?

Benefit	**Description**
Readability	Easy to understand how object is created from different formats
Flexibility	You can have multiple ways to construct an object
Encapsulation	Keep initialization logic inside the class

Key Point:

- `@classmethod` receives the **class (`cls`)** as the first argument, not the instance (`self`).
- It can **return a new object** by calling `cls(...)`, i.e., the main constructor.

1. `dir()` – Introspection Function:

Returns a list of attributes and methods associated with an object.

Example:

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

p = Person("Neeraj")
print(dir(p))
```

Output (partial):

```
['__class__', '__delattr__', '__dict__', 'name', '__init__', ...]
```

Key Points:

- Returns both user-defined and special methods.
- If used without arguments, it shows names in the current local scope.
- Useful for exploration/debugging.

TECHVISION TECHNOLOGY

2. __dict__ – Instance Attribute Dictionary

Returns a dictionary containing all instance variables (i.e., object's attributes).

Example:

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

s = Student("Neeraj", 23)
print(s.__dict__)
```

Output:

```
{'name': 'Neeraj', 'age': 23}
```

Key Points:

- Shows only **data attributes** of the object, not methods.
- Returns a dictionary, so you can access, update, or inspect attributes.
- Commonly used in debugging and serialization (like JSON conversion).

3. help() – Documentation Function

Displays the help/documentation for a module, class, method, or function.

Example:

```
help(str.upper)
```

Output:

```
Help on built-in function upper:
upper(...)
    Return a copy of the string converted to uppercase.
```

Key Points:

- Shows what a function/class does.
- Very useful to understand unfamiliar functions/modules.
- Best used in interactive environments like Python shell or Jupyter Notebook.

TECHVISION-TECHNOLOGY

`Super()` Keyword in Python:

- `super()` is a built-in function used to call the **parent class's methods or constructor**.
- It is mostly used in **inheritance** to reuse code from the parent class.

Why use `super()`

- To reuse parent class code (constructor or methods)
- To maintain clean and DRY (Don't Repeat Yourself) code
- Helps in multiple inheritance by following Method Resolution Order (MRO)

Example 1: Using `super()` to Call Parent Constructor

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

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)   # calls Person's
constructor
        self.student_id = student_id
```

Example 2: Calling Parent Method Using `super()`

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

    def show_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def show_info(self):
        super().show_info()  # call parent method
        print(f"Student ID: {self.student_id}")
```

Key Points:

- super() automatically refers to the immediate parent class.
- You can use super() to call any method, not just __init__.
- Works well with method overriding — lets you extend parent functionality.

Magic Dunder Methods:

Magic Dunder Methods in Python (also known as **"dunder methods"** because of the double underscores before and after their names like `__init__`, `__str__` are **special methods** that allow you to define how your objects behave with built-in Python operations.

Here's a structured list with common dunder methods and their uses

TECHVISION-TECHNOLOGY

1. Object Initialization & Representation

Method	**Purpose**	**Example**
`__init__(self, ...)`	Constructor	Called when object is created
`__new__(cls, ...)`	Creates a new instance (used rarely)	Before `__init__`
`__str__(self)`	User-friendly string (used in `print()`)	`print(obj)`
`__repr__(self)`	Official string, used in debugging	`repr(obj)`

Example:

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

    def __str__(self):
        return f"My name is {self.name}"

    def __len__(self):
        return len(self.name)

p = Person("Neeraj")
print(p)            # My name is Neeraj
print(len(p))       # 6
```

Operator Overloading in Python

Operator Overloading means giving **custom meaning to operators (+, -, , etc.)* for **user-defined objects** (i.e., class instances).

Python allows you to **override special methods** (also called **dunder methods**) like `__add__`, `__sub__`, `__mul__`, etc., to **define behavior of operators** for your objects.

Why Use It?

To perform **intuitive operations** on custom objects, like:
`obj1 + obj2`   # Instead of calling obj1.add(obj2)

Common Dunder Methods for Operator Overloading

TECHVISION-TECHNOLOGY

Operator	Method	Example
+	`__add__`	`obj1 + obj2`
-	`__sub__`	`obj1 - obj2`
*	`__mul__`	`obj1 * obj2`
/	`__truediv__`	`obj1 / obj2`
==	`__eq__`	`obj1 == obj2`
>	`__gt__`	`obj1 > obj2`
<	`__lt__`	`obj1 < obj2`

Example 1: Operator Overloading with +

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __add__(self, other): # Overloading +
    return Point(self.x + other.x, self.y + other.y)

  def __str__(self):
    return f"({self.x}, {self.y})"

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2 # This calls p1.__add__(p2)
print(p3)  # Output: (6, 8)
```

Example 2: Overloading == Operator:

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

  def __eq__(self, other): # Overloading ==
    return self.marks == other.marks

s1 = Student("Neeraj", 90)
s2 = Student("Amit", 90)
```

TECHVISION-TECHNOLOGY

```
print(s1 == s2) # Output: True
```

Example 3: Overloading > Operator

```
class Box:
  def __init__(self, volume):
    self.volume = volume

  def __gt__(self, other): # Overloading >
    return self.volume > other.volume

b1 = Box(100)
b2 = Box(80)

print(b1 > b2) # Output: True
```

Important Points:

- Operator overloading is done using **dunder (double underscore) methods**.
- The method is automatically called when the corresponding operator is used.
- Both **objects involved** should be of compatible types.

TECHVISION TECHNOLOGY

Single Level Inheritance in Python

Single Level Inheritance means that **a child class inherits from a single parent class**.
Child class gets access to all the **methods and properties** of the parent class.

Example 1: With Constructor and **`super()`**

```
class Person:
  def __init__(self, name):
    self.name = name
    print(f"Person created: {self.name}")

class Student(Person):
  def __init__(self, name, roll):
    super().__init__(name) # Call parent constructor
    self.roll = roll
    print(f"Student created: Roll No {self.roll}")

s = Student("Neeraj", 101)
```

Output:

```
Person created: Neeraj
Student created: Roll No 101
```

Key Points:

- In single level inheritance:
- One **parent** → One **child**

- Child can **override** parent methods.
- Use `super()` to call parent constructor or methods.
- Helps in **code reusability**.

Diagram (Text Form):

```
     Parent
       ↑
      Child
```

Use Cases:

- Reuse common functionality (e.g., login system, user info).
- Add specific behaviour in child class (e.g., student, employee).

What is Multiple Inheritance?

Multiple inheritance means a class can **inherit from more than one parent class**.

Example:

```
class Father:
    def skills(self):
        print("Father: Cooking")

class Mother:
    def skills(self):
        print("Mother: Painting")

class Child(Father, Mother):
    def skills(self):
        print("Child: ", end="")
        super().skills()
# Will call Father's skills() because Father is listed first

obj = Child()
obj.skills()
```

Output:

```
Child: Cooking
```

Important Notes:

- Python **resolves conflicts using MRO** (Method Resolution Order).
- The method of the first parent (from left to right) is called if there is a name conflict.
- To see MRO of a class, use:

```
print(Child.__mro__)
```

Using All Parents' Methods:
If you want to call methods from **all parent classes**, do it explicitly:

TECHVISION-TECHNOLOGY

```
class Child(Father, Mother):
    def skills(self):
        Father.skills(self)
        Mother.skills(self)
```

Advantages:

- Allows **code reuse** from multiple sources.
- Good for combining features from multiple classes.

Disadvantages:

- Can become **confusing** when multiple parents have the **same method name** (conflict).
- Can lead to **complex dependency structure**.

What is Multilevel Inheritance:

Multilevel inheritance means that a class **inherits from a child class**, which in turn **inherits from another parent class**.

It creates a **chain of inheritance** like:
Grandparent → Parent → Child

Example:

```
class Grandfather:
    def show_grandfather(self):
        print("I am the Grandfather.")

class Father(Grandfather):
    def show_father(self):
        print("I am the Father.")

class Son(Father):
    def show_son(self):
        print("I am the Son.")

# Create object of the last class in the chain
obj = Son()

# Access all methods
obj.show_grandfather()
obj.show_father()
obj.show_son()
```

Output:

```
I am the Grandfather.
I am the Father.
I am the Son.
```

TECHVISION-TECHNOLOGY

Key Points:

- The child class gets access to **all features** of parent and grandparent classes.
- It promotes **code reuse** across multiple levels.
- Python allows any number of levels in the inheritance chain.

Advantages:

- Helps in building a clear hierarchy.
- Promotes **step-by-step inheritance** of features.

Disadvantages:

- If the chain becomes too long, it can become **complex to manage**.
- Changes in the top-level class may affect all subclasses.

What is Hybrid Inheritance

Hybrid inheritance is a combination of **two or more types of inheritance** (like single, multiple, multilevel, hierarchical) in a single program.

It creates a **complex structure** where a class can inherit from multiple classes in different ways.

Example Structure:

Imagine this:

```
     A
    / \
   B   C
    \ /
     D
```

Here:

- B and C inherit from A (Hierarchical)
- D inherits from both B and C (Multiple)

This combination is known as **Hybrid Inheritance**.

Code Example:

```
class A:
    def feature_a(self):
        print("Feature A")

class B(A):
    def feature_b(self):
        print("Feature B")

class C(A):
    def feature_c(self):
        print("Feature C")
```

TECHVISION-TECHNOLOGY

```
class D(B, C):  # Inherits from both B and C
    def feature_d(self):
        print("Feature D")

# Create object of class D
obj = D()

# Accessing features from all parent classes
obj.feature_a()
obj.feature_b()
obj.feature_c()
obj.feature_d()
```

Output:

```
Feature A
Feature B
Feature C
Feature D
```

Key Concepts:

- Python uses **MRO (Method Resolution Order)** to resolve conflicts in Hybrid Inheritance.
- You can check the order using:

```
print(D.__mro__)
```

Advantages:

- Allows flexible and **powerful class design**.
- Enables combining multiple features from different class hierarchies.

Disadvantages:

- Can be **difficult to manage and understand**, especially if class names or method names conflict.
- Increases **complexity** of the codebase.

Summary:

- Hybrid Inheritance = **Mix of multiple inheritance types**.
- Use it carefully to avoid confusion.
- Python handles it smartly using MRO (left-to-right order).

TECHVISION-TECHNOLOGY