OOPS:

Class: blue print

Object: instance of the class

**Inheritance:**

* **Code reusability**
* **Easy readability and debugging**

**Super() method to access methods that are defined in the superclass in the subclass, i.e. basically overcoming the method overriding.**

**Syntax:** super().method\_name(args)

**Note:** Here we are using this Super() method in the child class method use the parent functionality in addition to the child functionality  
# Not after the object is created  
Example 1:

class Person:

def \_\_init\_\_(self, name):

self.name = name

class Employee(Person):

def \_\_init\_\_(self, name, emp\_id):

super().\_\_init\_\_(name) # calls Person.\_\_init\_\_

self.emp\_id = emp\_id

Example-2:  
class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

super().speak() # Calls Animal's speak()

print("Dog barks")

**Method Resolution Order:**

**If two superclasses have the same method (**[**function**](https://www.programiz.com/python-programming/function)**) name and the derived class calls that method, Python uses the MRO to search for the right method to call. For example,**

**class SuperClass1:**

**def info(self):**

**print("Super Class 1 method called")**

**class SuperClass2:**

**def info(self):**

**print("Super Class 2 method called")**

**class Derived(SuperClass1, SuperClass2):**

**pass**

**d1 = Derived()**

**d1.info()**

**# Output: "Super Class 1 method called"**

**In this case, the MRO specifies that methods should be inherited from the leftmost superclass first, so info() of SuperClass1 is called rather than that SuperClass2.**

**Self.variable\_name vs variable\_name in the OOPS:**

self.variable\_name is used when we need this variable to be used in the entire class and different methods of the class. Meaning in a way it gives the global access within the class. Whereas variable\_name is used and accessible only within that particular method of the class where it has been defined.

**Class**.**variable**\_**name vs self.variable\_name**

**# Class variables are known as static variables, static methods**

Lets look at the following example

class Car:

wheels = 4 # Class variable (shared across all instances)

def \_\_init\_\_(self, brand):

self.brand = brand # Instance variable (unique to each object)

def show\_info(self):

print(f"Car: {self.brand}, Wheels: {Car.wheels}")

# \*\*Both cars share the same `wheels` value\*\*

car1 = Car("Toyota")

car2 = Car("Honda")

car1.show\_info() # Output: Car: Toyota, Wheels: 4

car2.show\_info() # Output: Car: Honda, Wheels: 4

# \*\*Modifying class variable affects all instances\*\*

Car.wheels = 6

car1.show\_info() # Output: Car: Toyota, Wheels: 6

car2.show\_info() # Output: Car: Honda, Wheels: 6

**Here in the above example, variable** wheels is defined in the class, and by default every instance of the class is has a value of wheels as 4.

Self.variable emphasis that the particular variable belongs to that particular instance of the class.

**Encapsulation:**

**Encapsulation** is the **bundling of data and methods** that operate on that data within a single unit — usually a class. It also **restricts direct access** to some of the object’s components. In simple terms, it **hides internal state** and only exposes necessary parts.

**✅ Why do we use it?**

* To **protect the data** from unauthorized access or modification.
* To maintain **control** over how the data is accessed or changed.
* Makes code **more modular, readable, and maintainable**

| **Modifier** | **Syntax** | **Meaning** |
| --- | --- | --- |
| Public | self.name | Accessible everywhere |
| Protected | self.\_name | Convention: treat as internal, accessed by child classes |
| Private | self.\_\_name | Name mangled, can't be accessed directly |

class BankAccount:

def \_\_init\_\_(self, owner, balance):

self.owner = owner # public

self.\_\_balance = balance # private

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

def get\_balance(self): # getter

return self.\_\_balance

def withdraw(self, amount):

if amount <= self.\_\_balance:

self.\_\_balance -= amount

else:

print("Insufficient funds")

acc = BankAccount("Alice", 1000)

acc.deposit(500)

print(acc.get\_balance()) # 1500

# acc.\_\_balance → Error (private)

**2. Getters and Setters**

**📘 Definition:**

**Getter** and **setter** methods are used to **read and modify private attributes** of a class, following encapsulation principles.

**🧲 3. Static Methods**

**📘 Definition:**

A **static method** is a method that belongs to a class, **not an instance**, and **does not access** or **modify** class or instance data.

* Defined using @staticmethod decorator.

**✅ When do we use them?**

* When a method **logically belongs to the class**, but **doesn’t use self or cls**.
* For **utility/helper functions** related to the class.
* We can access through class
* Classname.static\_method()

class MathUtils:

@staticmethod

def add(a, b):

return a + b

@staticmethod

def is\_even(num):

return num % 2 == 0

print(MathUtils.add(3, 5)) # 8 #accesed through class name

print(MathUtils.is\_even(10)) # True

**🔁 What is Polymorphism?**

**📘 Definition:**

**Polymorphism** means **“many forms.”**  
In Object-Oriented Programming (OOP), polymorphism allows **objects of different classes** to be **treated as if they were of the same class**, because they share the same method name or interface.

It enables **methods with the same name** to behave **differently** depending on the object that calls them.

**✅ Why do we need Polymorphism?**

* Helps in **code reusability**.
* Simplifies code via a **common interface**.
* Supports **extensibility** in programs (add new types easily).
* Makes your code **more flexible and scalable**.

**🧠 Types of Polymorphism in Python**

| **Type** | **Description** |
| --- | --- |
| **Duck Typing** | Based on behavior (not type) |
| **Method Overriding** | Subclass redefines a method of parent class |
| **Operator Overloading** | Same operator behaves differently depending on operands |
| **Function Polymorphism** | Built-in functions that work on different data types |

Examples: **Duck Typing**

class Dog:

def speak(self):

return "Woof!"

class Cat:

def speak(self):

return "Meow!"

def animal\_sound(animal):

print(animal.speak())

# Duck typing in action

dog = Dog()

cat = Cat()

animal\_sound(dog) # Woof!

animal\_sound(cat) # Meow!

**animal\_sound() doesn’t care what type animal is, just that it has a .speak() method.**

**Method Overriding:**

class Vehicle:

def move(self):

print("Vehicle is moving")

class Car(Vehicle):

def move(self):

print("Car is driving")

class Boat(Vehicle):

def move(self):

print("Boat is sailing")

# Polymorphism through overriding

vehicles = [Car(), Boat()]

for v in vehicles:

v.move()

output:

Car is driving

Boat is sailing

**➕ 3. Operator Overloading**

You can define how **operators like +, \*, ==** behave for custom objects.

**🔁 4. Function Polymorphism (Built-in)**

Many built-in functions are polymorphic:

**✅ Example:**

python

print(len("Karthikeya")) # 10 (string)

print(len([1, 2, 3])) # 3 (list)

print(len({"a": 1, "b": 2})) # 2 (dict)

✅ len() works on **different types** — polymorphism in action.

**Some Important Pointers:**

* 1. Everything in python is an object. List, Str, Tuple etc… Hence Python is object-oriented programming language.
  2. Object behavior is same as that of any other variable in python. We can pass the object as an argument to a function, we can return obj as a result from a function.
  3. Objects of a class are mutable like lists, dict, sets. Hence we can assign a new variable for a class using object of that class. Note: this variable will only be valid for that object.
     + Example: obj=Animal()

Obj.name=”Dog”

* 1. We can call defined methods of a call under constructor.
  2. Nothing is truly private in python. You can access private methods, attributes using obj.\_classname\_\_privatemethod
  3. Get and Set methods are used to access and assign the values for private methods/attributes
  4. Class variable – variables which are same for all instances of that class.
  5. **Static methods and class variables** are accessed using class name.
     + **@static method**
  6. Method Resolution Order-> Multiple Inheritance: MRO tells python implements the method specified in the left most parent class.
     + class DerivedClass(Parent1, Parent2):
     + Obj=DerivedClass()
     + Obj.method() ----> this executes the Parent1 Implementation.
  7. Method overriding: Child class implements the method specified in child, overrides the parent implementation. Super() function can be used to implement parent methods in child methods.