✅ Generators in Python — Complete Explanation with Examples

🔷 What is a Generator?

A **generator** in Python is a special type of iterator that allows you to iterate through a sequence of values **lazily**, meaning it generates items **one at a time** only when needed.

🔷 Key Features:

- Uses **`yield`** instead of `return`.
- Automatically remembers its state between calls.
- More **memory-efficient** for large datasets.

🔷 Why Use Generators?

- To handle **large data** without loading everything into memory.
- To **improve performance**.
- To implement **pipelines** for data processing.

🔷 How to Create a Generator

1. Using a Function with `yield`

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

for value in gen:
    print(value)
```

Output:

```
1
2
3
```

---

2. Using Generator Expressions (like list comprehensions)

```
gen = (x*x for x in range(5))

for val in gen:
    print(val)
```

Output:

```
0
1
4
9
16
```

◈ Difference Between `return` and `yield`

Feature	`return`	`yield`
Terminates function	Yes	No, it pauses
Returns	A single value	A generator object
Use case	Regular functions	Iterators / streams of data

◈ Real-time Example: Fibonacci Series Generator

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

for num in fibonacci(10):
    print(num)
```

Output:

```
0
1
1
2
3
5
8
```

◈ Checking Generator Type

```
def sample():
    yield 1

print(type(sample()))  # <class 'generator'>
```

◈ When to Use a Generator?

- Reading large files line by line
- Streaming data (e.g., logs, sockets)
- Implementing infinite sequences (e.g., Fibonacci, prime numbers)

✅ 1. Reading a Large File Line by Line

```
def read_large_file(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()

for line in read_large_file('bigdata.txt'):
    print(line)
```

Use case: Useful for processing huge log files or datasets without loading the entire file into memory.

✅ 2. Sensor Data Simulation

```
import random
import time

def temperature_sensor():
    while True:
        yield round(random.uniform(20.0, 30.0), 2)
        time.sleep(1)

sensor = temperature_sensor()

for _ in range(5):
    print(next(sensor))
```

Use case: Simulate real-time sensor readings (e.g., IoT device streaming data).

✅ 3. Pagination Example (Chunking Data)

```
def get_items_in_pages(items, page_size):
    for i in range(0, len(items), page_size):
        yield items[i:i+page_size]

data = list(range(1, 21))  # 20 items

for page in get_items_in_pages(data, 5):
    print("Page:", page)
```

Use case: Split long lists into smaller chunks for pagination in web apps or reports.

✅ 4. Countdown Timer Generator

```
import time

def countdown(seconds):
    while seconds > 0:
        yield seconds
        seconds -= 1
        time.sleep(1)

for sec in countdown(5):
    print(f"Time left: {sec} seconds")
```

Use case: Building a timer or countdown feature in a game or productivity tool.

✅ 5. Streaming Prime Numbers (infinite generator)

```
def generate_primes():
    num = 2
    while True:
        for i in range(2, num):
            if num % i == 0:
                break
        else:
            yield num
        num += 1

prime_gen = generate_primes()

for _ in range(10):
    print(next(prime_gen))
```

Use case: Infinite sequence generation — ideal for algorithm testing or simulations.

1. What is a Decorator?

A **decorator** in Python is a **function that modifies the behavior of another function** or method without changing its code.

It's a **higher-order function** (a function that takes another function as input and returns a function).

---

2. Function as First-Class Object

In Python, functions are **first-class citizens**. This means:

- Functions can be passed as arguments.
- Functions can be returned from other functions.
- Functions can be assigned to variables.

Example:

```
def greet():
    return "Hello"

hello = greet  # assign function to a variableprint(hello())  # prints "Hello"
```

---

3. Basic Decorator Syntax

```
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
```

```
    return wrapper
@my_decoratordef say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Before function call
Hello!After function call
```

---

4. How Decorators Work (Flow Diagram)

```
Original Function
      ↓
 Decorator Function
      ↓
 Wrapper Function
      ↓
Modified Output
```

---

5. Using Decorators Without @ Syntax

```
def say_hi():
    print("Hi!")

decorated = my_decorator(say_hi)
decorated()
```

---

6. Decorator with Arguments

To handle arguments, use `*args` and `**kwargs`:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Function is being called with:", args, kwargs)
        return func(*args, **kwargs)
    return wrapper
@my_decoratordef add(x, y):
    return x + y
print(add(5, 3))
```

---

7. Real-Time Use Cases

✅ Logging

```
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper
@logdef multiply(a, b):
    return a * b
```

✅ Execution Time Tracker

```
import time
def timer(func):
    def wrapper(*args, **kwargs):
```

```
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} executed in {end-start:.4f} sec")
        return result
    return wrapper
@timerdef slow_task():
    time.sleep(2)
    print("Task done")
```

✅ **Authentication**

```
def authenticate(func):
    def wrapper(user):
        if user.get("logged_in"):
            return func(user)
        else:
            return "Access Denied"
    return wrapper
@authenticatedef show_dashboard(user):
    return f"Welcome {user['name']}"

user = {"name": "Alice", "logged_in": True}print(show_dashboard(user))
```

---

8. Nesting Multiple Decorator

```
def decorator1(func):
    def wrapper():
```

```
        print("Decorator 1")
        return func()
    return wrapper
def decorator2(func):
    def wrapper():
        print("Decorator 2")
        return func()
    return wrapper
@decorator1@decorator2def say_hi():
    print("Hi!")

say_hi()
```

Output:

```
Decorator 1
Decorator 2
Hi!
```

---

9. Class-Based Decorator

```
class Logger:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"Calling {self.func.__name__}")
```

```
        return self.func(*args, **kwargs)
@Loggerdef greet(name):
    print(f"Hello {name}")

greet("Tom")
```

---

10. Built-in Decorators in Python

Decorator	Description
@staticmethod	Declares a static method
@classmethod	Declares a class method
@property	Converts method to read-only property

```
class Circle:
    def __init__(self, radius):
        self._radius = radius
    @property
    def area(self):
        return 3.14 * self._radius ** 2

c = Circle(5)print(c.area)
```

---

11. `functools.wraps` to Preserve Metadata

```
from functools import wraps
def my_decorator(func):    @wraps(func)
```

```
    def wrapper(*args, **kwargs):
        print("Calling function")
        return func(*args, **kwargs)
    return wrapper
@my_decoratordef say_hi():
    """This function says hi"""
    print("Hi")
print(say_hi.__name__)  # say_hiprint(say_hi.__doc__)   # This function says hi
```

Example:

```
def admin_required(func):
    def wrapper(user):
        if user.get("role") == "admin":
            return func(user)
        else:
            return "Access Denied. Admins only."
    return wrapper

@admin_required
def delete_user(user):
    return f"{user['name']}'s account deleted."

print(delete_user({"name": "Alice", "role": "admin"}))  # ✅ Allowed
print(delete_user({"name": "Bob", "role": "user"}))     # ❌ Denied
```

✅ Summary Table

Feature	Description
Decorator	Function that modifies another function
Syntax	@decorator_name
With arguments	Use *args, **kwargs
Use cases	Logging, security, timing, validation
Nested decorators	Multiple decorators can wrap a single function
functools.wraps	Preserves metadata of original function

Classes and Objects in Python

✅ What is a Class?

A **class** is a blueprint for creating objects. It defines attributes (variables) and behaviors (methods) that the created objects will have.

✅ What is an Object?

An **object** is an instance of a class. When a class is defined, no memory is allocated. When it is instantiated (i.e., an object is created), memory is allocated.

Syntax of Class and Object

```
class ClassName:

    # constructor

    def __init__(self, attribute1, attribute2):

        self.attribute1 = attribute1

        self.attribute2 = attribute2
```

```python
    # method
    def display(self):
        print("Attribute 1:", self.attribute1)
        print("Attribute 2:", self.attribute2)
# creating an object
obj = ClassName("value1", "value2")
obj.display()
```

Real-Time Example: Student Class

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

    def show_details(self):
        print(f"Student Name: {self.name}")
        print(f"Roll Number: {self.roll}")
# creating objects
s1 = Student("Amit", 101)
s2 = Student("Sara", 102)

s1.show_details()
s2.show_details()
```

Key Concepts

Concept	Explanation
`__init__()`	Constructor method called when object is created.
self	Refers to the current instance of the class.
Methods	Functions defined inside a class.
Attributes	Variables associated with an object.

Example: Bank Account

```
class BankAccount:
    def __init__(self, holder_name, balance=0):
        self.holder_name = holder_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. New Balance: {self.balance}")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"{amount} withdrawn. New Balance: {self.balance}")
        else:
            print("Insufficient Balance")
# Create account object
account = BankAccount("John Doe", 1000)
```

```
account.deposit(500)

account.withdraw(700)

account.withdraw(1000)
```

✅ Summary

- **Class** = Template/Blueprint
- **Object** = Instance of a class
- Use `__init__` to initialize object values.
- Access object data using `self`.

Methods:

In Python, methods inside a class can be of three types:

1. **Instance Methods**
2. **Class Methods**
3. **Static Methods**

---

1. Instance Method

An instance method operates on the **object instance**. It can access and modify **instance variables**.

✅ Syntax:

```
class MyClass:
    def instance_method(self):
        print("This is an instance method")
```

- Takes `self` as the **first parameter**
- Can access instance variables via `self`

✅ Example:

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

    def greet(self):
        print(f"Hello, my name is {self.name}")

p1 = Person("Alice")
p1.greet()
```

---

2. Static Method

A static method **does not depend on object instance** or class variables. It behaves like a regular function but belongs to the class's namespace.

✅ Syntax:

```
class MyClass:    @staticmethod
    def static_method():
        print("This is a static method")
```

Marked with `@staticmethod`

Takes **no** `self` **or** `cls` parameter

Cannot access or modify instance/class variables

✅ Example:

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

result = MathUtils.add(10, 20)
print("Sum:", result)
```

Difference Table

Feature	Instance Method	Static Method
Decorator	None	@staticmethod
First Parameter	self	No self or cls
Access to Instance	✅ Yes	❌ No
Access to Class	✅ (via self/class)	❌ No
Use Case	For object behavior	Utility/helper functions

Example with Both

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

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

    def show_name(self):  # Instance method
        print("Student Name:", self.name)
    @staticmethod
    def school_info():  # Static method
        print("School is open 9 AM to 3 PM")

s1 = Student("Ravi")
s1.show_name()          # Instance method
Student.school_info()   # Static method
```

✅ Class Method in Python

A **class method** is a method that operates on the **class itself**, not on an instance. It is used when you want to **access or modify class-level data** (i.e., variables shared by all instances).

Key Points

Feature	Description
Decorator	@classmethod

Feature	Description
First parameter	cls (refers to the class, not object)
Access to class vars	✅ Yes
Access to instance vars	❌ No

✅ Syntax

```
class ClassName:    @classmethod
    def method_name(cls):
        # access class variables using cls
```

Real-Time Example: Factory Method for Students

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

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

    def show(self):  # Instance Method
        print(f"{self.name} studies in grade {self.grade} at
{Student.school_name}")
    @classmethod
    def change_school(cls, new_name):
```

```python
        cls.school_name = new_name
```

✅ **Usage:**

```python
# Create instances
s1 = Student("Amit", 5)
s2 = Student("Sara", 6)

s1.show()
s2.show()
# Change school name using class method
Student.change_school("Global High School")

s1.show()
s2.show()
```

Output:

```
Amit studies in grade 5 at ABC International School
Sara studies in grade 6 at ABC International School
Amit studies in grade 5 at Global High School
Sara studies in grade 6 at Global High School
```

When to Use `@classmethod`?

To access/modify **class variables**

To define **factory methods** that create objects in a specific way

---

Example: Factory Method

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    @classmethod
    def from_string(cls, emp_str):
        name, salary = emp_str.split("-")
        return cls(name, int(salary))

emp1 = Employee.from_string("John-50000")print(emp1.name)      # 
Johnprint(emp1.salary)   # 50000
```

✅ Method Overloading in Python

Method Overloading means having **multiple methods with the same name but different parameters**. In many languages like Java or C++, this is done explicitly.

But in Python, true method overloading is **not supported directly** because Python **does not support function/method signatures** like Java or C++.
However, we can **simulate** method overloading using:

Default arguments

Variable-length arguments (`*args`, `**kwargs`)

---

1. Using Default Arguments

```
class Greet:
    def hello(self, name=None):
        if name:
            print(f"Hello {name}")
        else:
            print("Hello there!")

g = Greet()
g.hello()          # Hello there!
g.hello("Amit")    # Hello Amit
```

---

2. Using `*args` for Variable Arguments

```
class Calculator:
    def add(self, *args):
        return sum(args)

c = Calculator()print(c.add(10, 20))          # 30print(c.add(5, 10, 15, 20))
# 50
```

Here, `add()` can accept **any number of arguments**, simulating method overloading.

---

3. Overriding `@singledispatchmethod` (Python 3.8+)

If you want real overloading behavior by type, you can use:

```
from functools import singledispatchmethod

class Printer:    @singledispatchmethod

    def show(self, arg):

        print(f"Default: {arg}")

    @show.register

    def _(self, arg: int):

        print(f"Integer: {arg}")

    @show.register

    def _(self, arg: str):

        print(f"String: {arg}")


p = Printer()

p.show(10)         # Integer: 10

p.show("Hello")    # String: Hello

p.show([1, 2, 3]) # Default: [1, 2, 3]
```

✅ @singledispatchmethod works like function overloading based on **type** of the first argument.

Summary

Technique	Description
Default parameters	Handle different cases with if checks
*args, **kwargs	Flexible number of arguments

Technique	Description
@singledispatchmethod	Real overloading based on argument type

✅ Method Overriding in Python

Method Overriding is a key concept of **Object-Oriented Programming (OOP)** where a **child class redefines a method** of its **parent class**.

What Is Method Overriding?

- Allows a **subclass** to provide a **specific implementation** of a method already defined in its **superclass**.
- Used when **child behavior is different** from parent behavior.

Syntax Example

```
class Parent:
    def show(self):
        print("This is the parent method")

class Child(Parent):
    def show(self):  # Overriding the parent method
        print("This is the child method")

# Usage
```

```
obj = Child()

obj.show()
```

Output:

```
pgsql

CopyEdit

This is the child method
```

✅ Real-Time Example: Bank Account

```
class Account:

    def interest_rate(self):

        print("Interest rate is 4%")

class SavingsAccount(Account):

    def interest_rate(self):  # Overriding

        print("Interest rate is 6% for savings account")


acc = SavingsAccount()

acc.interest_rate()
```

Output:

```
Interest rate is 6% for savings account
```

Calling Parent Method in Overridden Method

You can still call the **parent's version** using `super()`:

```
class Animal:

    def speak(self):
```

```
        print("Animal speaks")
class Dog(Animal):
    def speak(self):
        super().speak()  # Call parent method
        print("Dog barks")


d = Dog()
d.speak()
```

Output:

```
Animal speaks
Dog barks
```

Key Differences: Overloading vs Overriding

Feature	Overloading	Overriding
Definition	Same method name, different params	Same method name in child class
Occurs In	Same class	Between parent and child class
Python Support	Simulated (*args, @singledispatch)	Fully supported using inheritance

---

✅ Summary

Method overriding is used to **customize or extend** behavior.

The method must have **same name and parameters**.

Use `super()` to call the parent class version if needed.

Inheritance in Python

Inheritance allows a class (child or derived class) to **acquire the properties and methods of another class** (parent or base class). It promotes **code reuse** and establishes **is-a** relationships.

Types of Inheritance in Python:

Type	Description	Example
Single	One child inherits one parent	class B(A)
Multilevel	Chain of inheritance	class C(B), class B(A)
Hierarchical	Multiple children inherit one parent	class B(A), class C(A)
Multiple	One child inherits from multiple parents	class C(A, B)
Hybrid	Combination of two or more above types	Mixed styles

Basic Syntax of Inheritance:

```
class Parent:
    def display(self):
        print("This is the Parent class")
class Child(Parent):
    def show(self):
        print("This is the Child class")

obj = Child()
obj.display()  # Inherited
obj.show()     # Own method
```

Example: Single Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")
class Dog(Animal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()
d.bark()
```

Multilevel Inheritance

```
class A:
    def showA(self):
        print("A class")
class B(A):
    def showB(self):
        print("B class")
class C(B):
    def showC(self):
        print("C class")
```

```
c = C()
c.showA()
c.showB()
c.showC()
```

Ex:

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

    def show_employee_info(self):
        print(f"Employee Name: {self.name}, ID: {self.emp_id}")

class Developer(Employee):
    def __init__(self, name, emp_id, language):
        super().__init__(name, emp_id)
        self.language = language

    def show_developer_info(self):
        print(f"Developer Language: {self.language}")

class TechLead(Developer):
    def __init__(self, name, emp_id, language, team_size):
```

```
        super().__init__(name, emp_id, language)
        self.team_size = team_size

    def show_techlead_info(self):
        print(f"Tech Lead of {self.team_size} developers")

    def show_full_info(self):
        self.show_employee_info()
        self.show_developer_info()
        self.show_techlead_info()
lead = TechLead("Sana", 2003, "Java", 5)
lead.show_full_info()
```

Hierarchical Inheritance

```
class Vehicle:
    def start(self):
        print("Starting vehicle")
class Car(Vehicle):
    def drive(self):
        print("Driving car")
class Bike(Vehicle):
    def ride(self):
        print("Riding bike")

c = Car()
```

```
b = Bike()
c.start(); c.drive()
b.start(); b.ride()
```

Multiple Inheritance

```
class Father:
    def gardening(self):
        print("Loves gardening")
class Mother:
    def cooking(self):
        print("Loves cooking")
class Child(Father, Mother):
    def playing(self):
        print("Loves playing")

c = Child()
c.gardening()
c.cooking()
c.playing()
```

Hybrid Inheritance

Scenario:

We model a software company structure with:

`Employee` (base class)

`Manager` (inherits from `Employee`)

`Developer` (inherits from `Employee`)

`TechLead` (inherits from both `Manager` and `Developer`) — *Hybrid Inheritance*

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

    def show_employee(self):
        print(f"Name: {self.name}, ID: {self.emp_id}")
class Manager(Employee):
    def __init__(self, name, emp_id, department):
        super().__init__(name, emp_id)
        self.department = department

    def show_manager(self):
        print(f"Manages Department: {self.department}")
class Developer(Employee):
    def __init__(self, name, emp_id, language):
        super().__init__(name, emp_id)
        self.language = language
```

```
    def show_developer(self):
        print(f"Specialized in: {self.language}")
class TechLead(Manager, Developer):
    def __init__(self, name, emp_id, department, language, team_size):
        Manager.__init__(self, name, emp_id, department)
        Developer.__init__(self, name, emp_id, language)
        self.team_size = team_size

    def show_techlead(self):
        print(f"Leads a team of {self.team_size} developers")

    def show_full_info(self):
        self.show_employee()
        self.show_manager()
        self.show_developer()
        self.show_techlead()
lead = TechLead("Arjun", 5005, "AI Research", "Python", 6)
lead.show_full_info()
```

Output:

```
Name: Arjun, ID: 5005Manages Department: AI ResearchSpecialized in:
PythonLeads a team of 6 developers
```

Important Concepts

1. `super()` Keyword

Used to call parent class methods/constructors.

```
class Parent:
    def __init__(self):
        print("Parent Constructor")
class Child(Parent):
    def __init__(self):
        super().__init__()
        print("Child Constructor")

c = Child()
```

2. Method Overriding

Child class can **override** methods of the parent class.

```
class Parent:
    def greet(self):
        print("Hello from Parent")
class Child(Parent):
    def greet(self):
        print("Hello from Child")

c = Child()
c.greet()  # Overrides Parent method
```

Real-Life Examples

Employee → Manager, Developer:

```
class Employee:

    def __init__(self, name, emp_id, salary):

        self.name = name

        self.emp_id = emp_id

        self.salary = salary


    def show_details(self):

        print(f"Name: {self.name}, ID: {self.emp_id}, Salary:
₹{self.salary}")


# Manager inherits from Employee

class Manager(Employee):

    def __init__(self, name, emp_id, salary, department):

        super().__init__(name, emp_id, salary)

        self.department = department


    def show_details(self):

        super().show_details()

        print(f"Role: Manager, Department: {self.department}")


# Developer inherits from Employee

class Developer(Employee):

    def __init__(self, name, emp_id, salary, programming_language):
```

```
        super().__init__(name, emp_id, salary)

        self.language = programming_language


    def show_details(self):

        super().show_details()

        print(f"Role: Developer, Language: {self.language}")

mgr = Manager("Alice", 1001, 90000, "HR")

dev = Developer("Bob", 1002, 75000, "Python")


mgr.show_details()

print("-----")

dev.show_details()
```

- `super().__init__()` is used to call the parent constructor.
- Both `Manager` and `Developer` reuse the `Employee` class logic and extend it.

`Shape` → `Circle`, `Rectangle`

`Account` → `SavingsAccount`, `CurrentAccount`

Assignments

Create a base class `Person`, and a derived class `Student` that inherits name and age.

Create a class `Shape` with area method and inherit it in `Circle` and `Rectangle`.

Demonstrate multiple inheritance with classes `Artist`, `Athlete`, and `Personality`.

Override a parent method in the child class using `super()` to also call parent logic.

✅ What is Encapsulation?

Encapsulation is one of the four fundamental OOP principles (others are inheritance, polymorphism, abstraction).
It means **wrapping data (variables) and methods into a single unit (class)** and **restricting direct access** to some of the class's internal components.

> In simple terms: **Protecting sensitive data** from being modified directly.

Why Use Encapsulation?

✅ Data protection (e.g., salary, password)

✅ Controlled access using methods

✅ Cleaner, modular code

✅ Hides internal implementation

How is Encapsulation Achieved in Python?

`_single_underscore`: **Protected** (convention only)

`__double_underscore`: **Private** (name mangling)

Example: Encapsulation in Python

```
class Employee:

    def __init__(self, name, salary):

        self.name = name               # Public

        self.__salary = salary         # Private


    def show_info(self):

        print(f"Employee Name: {self.name}")

        print(f"Salary: ₹{self.__salary}")


    def update_salary(self, amount):

        if amount > 0:

            self.__salary = amount

        else:

            print("Invalid salary update!")


emp = Employee("Riya", 50000)

emp.show_info()


# Trying to access private variableprint(emp.name)             # ✓ Works#
print(emp.__salary)        ✗ AttributeError

# Correct way to update private field

emp.update_salary(60000)

emp.show_info()
```

❗ Can we still access private variables?

Yes, but it’s **not recommended** (Python allows it via name mangling):

```
print(emp._Employee__salary)  # Not good practice
```

Real-Life Examples:

Class	Encapsulated Data	Why Encapsulated?
BankAccount	__balance	Prevents unauthorized updates
Student	__marks, __id	Data security, controlled access
User	__password	Security, hash/encrypt password

Encapsulation with Getter/Setter (Manual)

```
class Student:
    def __init__(self):
        self.__grade = None

    def set_grade(self, g):
        if g in ['A', 'B', 'C']:
            self.__grade = g
        else:
            print("Invalid grade!")

    def get_grade(self):
```

```
        return self.__grade

s = Student()
s.set_grade('B')print(s.get_grade())
```

✅ Summary

Modifier	Syntax	Access
Public	self.name	Anywhere
Protected	_self.name	Convention (use within class & subclass)
Private	__self.name	Within class only (uses name mangling)

Banking system:

```
class BankAccount:
    def __init__(self, acc_holder, initial_balance):
        self.acc_holder = acc_holder
        self.__balance = initial_balance  # private

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"₹{amount} deposited.")
        else:
            print("Invalid deposit amount.")
```

```
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"₹{amount} withdrawn.")
        else:
            print("Insufficient balance or invalid amount.")

    def show_balance(self):
        print(f"Current balance: ₹{self.__balance}")

# ✅ Usage
acc = BankAccount("Rahul", 10000)
acc.show_balance()
acc.deposit(5000)
acc.withdraw(2000)
acc.withdraw(20000)  # invalid
acc.show_balance()

# Trying to access private variable (not recommended)
# print(acc.__balance)        ❌ AttributeError
print(acc._BankAccount__balance)  #    Not good practice
```

Login system:

```
class LoginSystem:
    def __init__(self, username, password):
```

```
        self.__username = username
        self.__password = password  # private

    def login(self, user, pwd):
        if self.__username == user and self.__password == pwd:
            print("✅ Login Successful")
        else:
            print("❌ Invalid credentials")

    def change_password(self, old_pwd, new_pwd):
        if self.__password == old_pwd:
            self.__password = new_pwd
            print("   Password changed successfully")
        else:
            print("❌ Incorrect old password")

# ✅ Usage
user1 = LoginSystem("admin", "pass123")

user1.login("admin", "wrongpass")    # ❌
user1.login("admin", "pass123")      # ✅

user1.change_password("wrong", "new")  # ❌
user1.change_password("pass123", "newpass")  # ✅

user1.login("admin", "newpass")      # ✅
```

- Used __ prefix to **encapsulate sensitive fields**.
- Methods like `login()`, `change_password()`, and `withdraw()` provide **controlled access**.
- Prevents direct manipulation of private variables.

✅ What is Abstraction?

Abstraction is an **Object-Oriented Programming (OOP)** concept that **hides internal implementation details** and **shows only essential features** to the user.

> In simple terms: **You use something without knowing how it works internally.**

Real-Life Example of Abstraction

ATM Machine: You withdraw cash without knowing how backend processes work.

Car: You drive using steering and pedals, without knowing the engine mechanics.

Python `len()` function: You use it without knowing its internal implementation.

How to Implement Abstraction in Python?

Python provides **abstraction** using the `abc` **module** (Abstract Base Classes).

Key tools:

`ABC`: Base class to mark abstract class

`@abstractmethod`: Decorator to define abstract methods

Example: Abstraction with ABC module

```
from abc import ABC, abstractmethod

class Vehicle(ABC):    @abstractmethod
    def start(self):
        pass
    @abstractmethod
    def stop(self):
        pass

class Car(Vehicle):
    def start(self):
        print("Car started with key")

    def stop(self):
        print("Car stopped")

# ✅ Usage
c = Car()
c.start()
c.stop()
```

❌ **If you try this:**

```
v = Vehicle()  # Error! Can't instantiate abstract class
```

Why Use Abstraction?

✅ Hide complex logic from user

✅ Enforce method definitions in child classes

✅ Design clear interfaces

Real-Time Use Case: Banking System

```
from abc import ABC, abstractmethod
class Account(ABC):    @abstractmethod
    def deposit(self, amount):
        pass
    @abstractmethod
    def withdraw(self, amount):
        pass
class SavingsAccount(Account):
    def __init__(self, balance):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ₹{amount}. New balance: ₹{self.balance}")

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew ₹{amount}. Remaining balance: ₹{self.balance}")
```

```
        else:
            print("Insufficient balance")
# ✅ Usage
acc = SavingsAccount(1000)
acc.deposit(500)
acc.withdraw(800)
```

Abstraction vs Encapsulation

Feature	Encapsulation	Abstraction
Purpose	Protect data from outside access	Hide implementation details
Achieved via	Private variables/methods	Abstract classes & methods
Focus	"How" data is protected	"What" is shown to the user
Example	__password in class	@abstractmethod in interface

Assignments

1. Create an abstract class `Appliance` with abstract methods `turn_on()` and `turn_off()`. Inherit it in `TV`, `WashingMachine`.
2. Design an abstract class `Shape` with `area()` method. Implement in `Circle`, `Rectangle`.
3. Build a mini **payment gateway** using an abstract class `Payment` with methods like `pay()`, and implement `UPIPayment`, `CardPayment`.

E-commerce Example using Abstraction

We'll simulate a basic **payment system** where different payment methods (like UPI, Credit Card, Wallet) follow a common interface but have different internal implementations.

```
from abc import ABC, abstractmethod
```

```python
# Abstract class for payment method
class PaymentMethod(ABC):

    @abstractmethod
    def pay(self, amount):
        pass

# Concrete class for UPI payment
class UPIPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} via UPI.")

# Concrete class for Credit Card payment
class CreditCardPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using Credit Card.")

# Concrete class for Wallet payment
class WalletPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} from Wallet.")

# Order class that takes a payment method (abstraction in action)
class Order:
    def __init__(self, payment_method: PaymentMethod):
        self.payment_method = payment_method

    def checkout(self, amount):
```

```
        print("Processing payment...")
        self.payment_method.pay(amount)
        print("✅ Payment successful!\n")

# ✅ Usage
upi = UPIPayment()
card = CreditCardPayment()
wallet = WalletPayment()

order1 = Order(upi)
order2 = Order(card)
order3 = Order(wallet)

order1.checkout(1200)
order2.checkout(2500)
order3.checkout(500)
```

What’s abstracted here?

- The `Order` class **doesn’t know** how the payment is processed.
- It **depends only on the abstract class** `PaymentMethod`.
- Each payment method defines its own `pay()` logic.

Advantages of Abstraction Here:

Easy to **add new payment types** (e.g., `NetBanking`, `Crypto`) without changing the `Order` class.

Promotes **interface-based design**.

Hides complex payment logic from the main application.

Magic Methods in Python

Magic methods (also known as **dunder methods**, because they have double underscores __ before and after their names) are special methods that define the behavior of objects for built-in operations.
They allow you to **customize how objects behave with operators, built-in functions, and type conversions**.

✅ Commonly Used Magic Methods

Magic Method	Purpose
__init__(self, ...)	Constructor - initializes a new object.
__str__(self)	Defines a human-readable string representation (print(object)).
__repr__(self)	Defines an official string representation (used in debugging).
__len__(self)	Defines behavior for len(object).
__getitem__(self, key)	Allows indexing like obj[key].
__setitem__(self, key, value)	Defines behavior for item assignment.
__delitem__(self, key)	Defines behavior for deleting an item with del.
__iter__(self)	Makes an object iterable.
__next__(self)	Defines iteration behavior for next().

Magic Method	Purpose
__call__(self, ...)	Makes an object callable like a function.
__eq__(self, other)	Defines behavior for equality ==.
__lt__(self, other)	Defines behavior for <.
__add__(self, other)	Defines behavior for + operator.
__sub__(self, other)	Defines behavior for - operator.
__del__(self)	Destructor – called when an object is deleted.

Python Magic Methods Cheat Sheet Magic methods (dunder methods) allow you to customize object behavior.

1. Object Creation & Initialization:

__new__(cls), __init__(self), __del__(self)

2. String Representation: __str__(self), __repr__(self)
3. Arithmetic Operators: __add__(self, other), __sub__, __mul__, __truediv__, __floordiv__, __mod__, __pow__
4. Comparison Operators: __eq__, __ne__, __lt__, __le__, __gt__, __ge__
5. Container Behavior: __len__, __getitem__, __setitem__, __delitem__, __contains__, __iter__, __next__
6. Callable Objects: __call__(self, *args, **kwargs)
7. Context Managers: __enter__, __exit__
8. Attribute Access: __getattr__, __setattr__, __delattr__, __dir__

9. Object Copying: __copy__, __deepcopy__ Example usage: class Point: def __init__(self, x): self.x = x def __add__(self, other): return Point(self.x + other.x) def __str__(self): return f"Point({self.x})"

Magic methods allow you to **customize object behavior** for built-in functions and operators. Here's a **complete categorized list** with examples.

✅ 1. Object Creation & Initialization

Method	Purpose
__new__(cls, ...)	Creates a new instance (rarely overridden).
__init__(self, ...)	Initializes the object (constructor).
__del__(self)	Destructor – called when an object is deleted.

```
class Demo:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print(f"{self.name} is deleted!")

obj = Demo("Test")del obj   # Output: Test is deleted!
```

✅ 2. String Representation

Method	Purpose
__str__(self)	Defines human-readable string (print(obj)).
__repr__(self)	Defines official string for debugging (repr(obj)).

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

```
    def __repr__(self):
        return f"Person('{self.name}')"

p = Person("Alice")print(p)      # Person: Aliceprint(repr(p))# Person('Alice')
```

✅ 3. Arithmetic Operators Overloading

Method	Operator
__add__(self, other)	+
__sub__(self, other)	-
__mul__(self, other)	*
__truediv__(self, other)	/
__floordiv__(self, other)	//
__mod__(self, other)	%
__pow__(self, other)	**

```
class Point:
    def __init__(self, x):
        self.x = x
    def __add__(self, other):
        return Point(self.x + other.x)
    def __str__(self):
        return f"Point({self.x})"
print(Point(3) + Point(7))    # Point(10)
```

✅ 4. Comparison Operators

Method	Operator
__eq__(self, other)	==
__ne__(self, other)	!=
__lt__(self, other)	<
__le__(self, other)	<=
__gt__(self, other)	>
__ge__(self, other)	>=

```
class Student:
    def __init__(self, marks):
        self.marks = marks
    def __gt__(self, other):
        return self.marks > other.marks
print(Student(80) > Student(70))  # True
```

✅ 5. Container Behavior (List/Dict-like Objects)

Method	Purpose
__len__(self)	Defines len(obj).
__getitem__(self, key)	Allows indexing obj[key].
__setitem__(self, key, value)	Allows item assignment.
__delitem__(self, key)	Allows deletion of items.
__contains__(self, item)	Defines behavior for in keyword.
__iter__(self)	Makes object iterable.
__next__(self)	Defines behavior for iteration using next().

```
class MyList:
    def __init__(self, data):
        self.data = data
    def __getitem__(self, index):
        return self.data[index]
    def __len__(self):
        return len(self.data)

nums = MyList([10, 20, 30])print(len(nums))   # 3print(nums[1])     # 20
```

✅ 6. Callable Objects

Method	Purpose
`__call__(self, *args, **kwargs)`	Makes an object callable like a function.

```
class Greet:
    def __call__(self, name):
        return f"Hello {name}!"

say = Greet()print(say("Alice"))   # Hello Alice!
```

✅ 7. Context Managers

Method	Purpose
`__enter__(self)`	Used with with statement (enter block).

Method	Purpose
__exit__(self, exc_type, exc_value, traceback)	Used with with (exit block).

```
class MyContext:
    def __enter__(self):
        print("Entering...")
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting...")
with MyContext():
    print("Inside block")
```

✓ 8. Attribute Access

Method	Purpose
__getattr__(self, name)	Called when attribute is not found.
__setattr__(self, name, value)	Called when setting an attribute.
__delattr__(self, name)	Called when deleting an attribute.
__dir__(self)	Customizes attributes shown by dir(obj).

✓ 9. Object Copying & Representation

Method	Purpose
__copy__(self)	Defines behavior for copy.copy().
__deepcopy__(self, memo)	Defines behavior for copy.deepcopy().

Module in Python

A **module** is a **file containing Python code** (functions, classes, or variables) that can be imported and reused in other programs.

- Modules make programs **modular**, **organized**, and **reusable**.
- Any Python file with `.py` extension is a module.

✅ Types of Modules in Python

1 Standard (Built-in) Modules

These come with Python installation. No need to install separately.

Examples:

- `math` – Mathematical operations
- `os` – Operating system interaction
- `datetime` – Date and time handling
- `sys` – System-related parameters
- `random` – Random number generation

Real-Time Example using Standard Modules

✅ Example 1: Using `math` for calculations

```
import math
```

```
radius = 5

area = math.pi * math.pow(radius, 2)print(f"Area of Circle: {area}")
```

Real-Time Use: Calculating areas, scientific computations in engineering applications.

✅ **Example 2: Using `datetime` in an attendance system**

```
from datetime import datetime


now = datetime.now()print("Login Time:", now.strftime("%Y-%m-%d %H:%M:%S"))
```

Real-Time Use: Logging user login times in attendance or employee management systems.

✅ **Example 3: Using `os` for file management**

```
import os

# Create a folder for reportsif not os.path.exists("Reports"):

    os.mkdir("Reports")print("Reports directory created!")
```

Real-Time Use: Automating folder creation for daily reports in a business application.

---

2 Custom (User-Defined) Modules

Custom modules are **Python files created by the user** to organize reusable code.

Creating and Using a Custom Module

mymodule.py (Custom Module)

```
def greet(name):
```

```
    return f"Welcome, {name}! You have successfully logged in."
def calculate_salary(basic, bonus):
    return basic + bonus

company_name = "Tech Solutions Pvt Ltd"
```

---

main.py (Using the Custom Module)

```
import mymodule
print(mymodule.greet("John"))    # Using a function from custom moduleprint("Total Salary:", mymodule.calculate_salary(40000, 5000))print("Company:", mymodule.company_name)
```

Real-Time Use:

A `billing_module.py` could have functions to calculate bills, taxes, discounts.

A `database_module.py` could handle database connections.

Real-Time Use: API calls in web applications.

✅ Difference Between Standard, Custom, and Third-Party Modules

Feature	Standard Modules	Custom Modules	Third-Party
Definition	Pre-installed with	User-created modules	Need to install

Feature	Standard Modules	Custom Modules	**Third-Party**
	Python		
Installation	No installation needed	No installation needed	
Examples	math, os, datetime	mymodule.py(user created module)	
Use Case	Common tasks	Project-specific code	

✅ Built-in Modules in Python (OS, Math, Sys, Datetime)

Python provides several **built-in modules** that simplify development by offering pre-written functionalities.
Here's a detailed explanation with **real-time examples** for `os`, `math`, `sys`, **and** `datetime`.

---

1. OS Module

The `os` module allows interaction with the **operating system** (file handling, directory management, environment variables, etc.).

✅ Common Functions:

- `os.getcwd()` → Get current working directory
- `os.listdir()` → List files/folders
- `os.mkdir("folder")` → Create new folder
- `os.remove("file.txt")` → Delete file
- `os.path.exists("path")` → Check if path exists

✅ Example:

```
import os
print("Current Directory:", os.getcwd())
# Create a folder if not existsif not os.path.exists("Logs"):
    os.mkdir("Logs")
    print("Logs folder created!")
```

Real-Time Use: Creating log folders, managing uploads in applications.

---

2. Math Module

The `math` module provides mathematical functions and constants.

✓ Common Functions:

- `math.sqrt(x)` → Square root
- `math.pow(x, y)` → x^y (power)
- `math.factorial(n)` → Factorial
- `math.pi` → π constant
- `math.ceil(x)` → Round up
- `math.floor(x)` → Round down

✓ Example:

```
import math

radius = 7
```

```
area = math.pi * math.pow(radius, 2)print("Area of Circle:", area)
```

Real-Time Use: Scientific calculations, financial modeling.

---

3. Sys Module

The `sys` module provides access to **system-specific parameters and functions**.

✅ **Common Functions:**

- `sys.version` → Python version
- `sys.exit()` → Exit program
- `sys.argv` → Command-line arguments
- `sys.path` → List of module search paths

✅ **Example:**

```
import sys

print("Python Version:", sys.version)

# Command-line argument example# Run as: python script.py Johnif
len(sys.argv) > 1:

    print("Hello,", sys.argv[1])
```

Real-Time Use: Handling command-line inputs, terminating scripts on errors.

---

4. Datetime Module

The `datetime` module is used for **working with dates and times**.

✅ **Common Functions:**

- `datetime.now()` → Current date & time `strftime("%Y-%m-%d")` → Format date

`timedelta(days=5)` → Date arithmetic

`date.today()` → Current date

✅ Example:

```
from datetime import datetime, timedelta

now = datetime.now()print("Current Time:", now.strftime("%Y-%m-%d %H:%M:%S"))
# Calculate date 7 days later
future_date = now + timedelta(days=7)print("After 7 days:",
future_date.strftime("%Y-%m-%d"))
```

Real-Time Use: Attendance systems, reminders, scheduling.

✅ Summary Table

Module	Purpose	Example Use Case
os	OS interaction (files, folders)	File management in apps
math	Mathematical functions/constants	Scientific apps, finance
sys	System parameters/functions	Command-line tools
datetime	Date & time operations	Logging, event scheduling