Python

Lambda Functions

- **What**: Anonymous (no name) functions defined using the `lambda` keyword.
- **Why**: Useful for short, one-line functions, often used in sorting, filtering, or mapping.

```
lambda arguments: expression
# Normal function
def add(a, b):
    return a + b

# Lambda function
add_lambda = lambda a, b: a + b

print(add(3, 5))         # 8
print(add_lambda(3, 5))  # 8

# Sort list of tuples by second element
pairs = [(1, 4), (2, 1), (3, 3)]
pairs.sort(key=lambda x: x[1])
print(pairs)  # [(2, 1), (3, 3), (1, 4)]

# Filter even numbers
nums = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)  # [2, 4]
```

Generators

- **What**: Functions that use `yield` instead of `return`.
- **Why**: Return values **one at a time** and remember their state between calls (lazy evaluation → memory efficient).
- **How**: Each call to `next()` resumes where it left off.

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

gen = my_generator()

print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
# print(next(gen))  # ❌ StopIteration error
```

`*args` – Variable Number of Positional Arguments

- **What**: Lets a function accept **any number** of **positional** arguments.
- **Type**: Stored as a **tuple**.
- **Use case**: When you don't know beforehand how many values will be passed.

```
def add_all(*args):
    print(args)  # tuple
    return sum(args)

print(add_all(1, 2, 3))         # 6
print(add_all(5, 10, 15, 20))   # 50
```

`**kwargs` – Variable Number of Keyword Arguments

- **What**: Lets a function accept **any number** of **keyword** arguments.
- **Type**: Stored as a **dictionary**.
- **Use case**: When you don't know beforehand which named arguments will be passed.

```
def print_details(**kwargs):
    print(kwargs)  # dict
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_details(name="Abhishek", role="SDET", location="India")
# Output:
# {'name': 'Abhishek', 'role': 'SDET', 'location': 'India'}
# name: Abhishek
# role: SDET
# location: India
```

Argument Unpacking

You can also **unpack** iterables/dictionaries when calling a function.

```
def show_info(name, age, city):
    print(name, age, city)

data_list = ["Abhishek", 25, "Delhi"]
data_dict = {"name": "Abhishek", "age": 25, "city": "Delhi"}

show_info(*data_list)   # unpack list → positional args
show_info(**data_dict)  # unpack dict → keyword args
```

List

- **Mutable** → can be changed after creation.
- **Ordered** → maintains insertion order (Python 3.7+ officially guarantees this).

- **Allows duplicates**.

Tuple

- **Immutable** → cannot be changed after creation.
- **Ordered**.
- **Allows duplicates**.

Set

- **Mutable** (can add/remove items).
- **Unordered** → no indexing.
- **No duplicates** (unique elements only).

```
my_set = {1, 2, 3, 2}
my_set.add(4)
print(my_set)       # {1, 2, 3, 4} (order not guaranteed)
```

Dictionary

- **Mutable**.
- **Ordered** (Python 3.7+).
- Stores **key–value** pairs.
- **Keys must be unique** (values can duplicate).

Feature	List	Tuple	Set	Dict
Mutable	✅	❌	✅	✅
Ordered	✅	✅	❌	✅
Duplicates	✅	✅	❌	Keys ❌ / Values ✅
Indexing	✅	✅	❌	By key
Syntax	`[ ]`	`( )`	`{ }`	`{key: value}`

List Comprehension

A concise way to create lists using a single line of code.

```
[expression for item in iterable if condition]
nums = [1, 2, 3, 4, 5]
squares = [n**2 for n in nums if n % 2 == 0]
print(squares)  # [4, 16]
```

Dict Comprehension

Similar idea, but for dictionaries — generates key-value pairs.

```
{key_expr: value_expr for item in iterable if condition}
nums = [1, 2, 3, 4, 5]
squares_dict = {n: n**2 for n in nums if n % 2 == 0}
print(squares_dict)  # {2: 4, 4: 16}
```

if else use-case:

```
labels = ["even" if n % 2 == 0 else "odd" for n in nums]
print(labels)  # ['odd', 'even', 'odd', 'even', 'odd']
```

Mutable objects → Can be changed after creation (contents can be modified in-place).

- **Immutable objects** → Cannot be changed after creation (any modification creates a new object).

```
# String (Immutable)
name = "Abhi"
print(id(name))     # memory address

name += "shek"      # creates a new string object
print(name)         # "Abhishek"
print(id(name))     # different address → new object created

# List (Mutable)
nums = [1, 2, 3]
print(id(nums))     # memory address

nums.append(4)      # modify in-place
print(nums)         # [1, 2, 3, 4]
print(id(nums))     # same address → changed in-place
```

Mutable objects can be changed without changing their identity.

- **Immutable objects** cannot be altered; operations create new objects.
- Be careful when using **mutable default arguments** in functions (can cause unexpected behavior).

```
def add_item(item, my_list=[]):  # Dangerous!
    my_list.append(item)
    return my_list

print(add_item(1))  # [1]
print(add_item(2))  # [1, 2] → same list reused!
```

In Python, **default argument values** are **evaluated only once —** at the time the function is **defined**, not each time it is called.

Opening a File

We use the built-in `open()` function:

open(file, mode)
`"r"` → Read (default)

- `"w"` → Write (overwrites file)
- `"a"` → Append (adds to end)
- `"x"` → Create new (fails if exists)
- `"b"` → Binary mode (e.g., `"rb"`, `"wb"`)
- `"t"` → Text mode (default, e.g., `"rt"`)

Reading

```
# Reading entire content
with open("example.txt", "r") as f:
    content = f.read()
    print(content)

# Reading line by line
with open("example.txt", "r") as f:
    for line in f:
        print(line.strip())

# Reading specific number of characters
with open("example.txt", "r") as f:
    part = f.read(10)  # reads first 10 chars
    print(part)

# Reading all lines into a list
with open("example.txt", "r") as f:
    lines = f.readlines()
    print(lines)
```

Writing

```
# Overwrite mode
with open("example.txt", "w") as f:
    f.write("Hello, World!\n")
    f.write("Second line\n")

# Append mode
with open("example.txt", "a") as f:
    f.write("This line is appended.\n")
```

Why use `with open()`

- It **automatically closes** the file (good practice, prevents memory leaks)
- Safer even if an exception occurs.

Json.load() = > load py dict from json file json.loads() => load py dict from json string

```
with open("data.json", 'r') as file:d
        data = json.load(file)
```

🕐 Use json.dumps() to convert Python objects into JSON strings.
 🕐 Use json.dump() to write Python objects into JSON files.

What is a Package?

A **package** is just a directory containing an `__init__.py` file

```
project/
│
├── math_utils/
│   ├── __init__.py
│   ├── addition.py
│   └── multiplication.py
│
└── main.py
```

```
# This file makes math_utils a package __init__.py
from .addition import add
from .multiplication import multiply

# Optional: define what gets imported with *
__all__ = ["add", "multiply"]
```

usecase: from math_utils import add, multiply

Random

```
import random

print(random.random())       # Random float between 0.0 and 1.0
print(random.randint(1, 10)) # Random integer between 1 and 10 (inclusive)
print(random.uniform(1, 5))  # Random float between 1 and 5

items = ["apple", "banana", "cherry"]
print(random.choice(items))  # Random element from a list
print(random.sample(items, 2))  # Random 2 unique elements
```

```
**import random
import string

def random_alpha(length=8):
    """Generate a random string of alphabets only."""
    return ''.join(random.choices(string.ascii_letters, k=length))

def random_numeric(length=8):
```

```
    """Generate a random string of digits only."""
    return ''.join(random.choices(string.digits, k=length))

def random_alphanumeric(length=8):
    """Generate a random string of letters and digits."""
    return ''.join(random.choices(string.ascii_letters + string.digits, k=length))


from datetime import datetime, timedelta

# 1. Convert date string to datetime object
date_str = "2025-08-15"  # YYYY-MM-DD format
date_format = "%Y-%m-%d"
date_obj = datetime.strptime(date_str, date_format)

print("Original datetime:", date_obj)

# 2. Add timedelta days
new_date_obj = date_obj + timedelta(days=5)

print("After adding 5 days:", new_date_obj)

# 3. Convert datetime back to string
new_date_str = new_date_obj.strftime(date_format)
print("New date string:", new_date_str)
```

Concurrency

- **Definition**: Doing **multiple tasks in overlapping time periods** — not necessarily at the exact same time.
- **Analogy**: Like a single chef cooking 3 dishes by switching between them — stir one pot, chop vegetables, then flip a pancake.
- **How it works**: Tasks share the same CPU core, switching rapidly (context switching).
- **Python example**: Using `asyncio` or multithreading (I/O-bound tasks).

Parallelism

- **Definition**: Doing **multiple tasks at exactly the same time**.
- **Analogy**: Having 3 chefs each cooking their own dish **at the same time** in separate kitchens.
- **How it works**: Tasks run on **multiple CPU cores** simultaneously.
- **Python example**: Using `multiprocessing` (CPU-bound tasks).

Feature	Concurrency	Parallelism
Goal	Manage multiple tasks efficiently	Speed up execution via simultaneity

Feature	Concurrency	Parallelism
Execution	Tasks overlap in time	Tasks run at the exact same time
Cores Needed	1 core is enough	Multiple cores required
Best For	I/O-bound tasks	CPU-bound tasks

Multithreading

- **Definition**: Multiple threads run in the **same process**, sharing the same memory space.
- **GIL Effect in Python**: Due to the Global Interpreter Lock, only **one thread** can execute Python bytecode at a time.
  → So in Python, multithreading **doesn't speed up CPU-bound tasks**.
- **Best For**: **I/O-bound tasks** (e.g., file I/O, network requests, API calls).
- **Memory Usage**: Low (shared memory space).
- **Overhead**: Low (lighter than processes).

Multiprocessing

- **Definition**: Multiple processes run independently, each with **its own Python interpreter** and memory space.
- **GIL Effect**: No restriction — each process has its own GIL.
  → **True parallelism** for CPU-bound tasks.
- **Best For**: **CPU-bound tasks** (e.g., heavy computation, data processing).
- **Memory Usage**: Higher (separate memory for each process).
- **Overhead**: Higher (process creation is costlier).

Iterable

- **Definition**: An object capable of returning its elements **one at a time**.
- Examples: **list, tuple, set, dict, string** — all are iterables.
- Technically: An iterable is any object that has an `__iter__()` method (or implements `__getitem__()` with sequential indexes).
- **Needs**: To be turned into an **iterator** to fetch elements one-by-one.

```
my_list = [1, 2, 3]  # This is an iterable
print(hasattr(my_list, '__iter__'))  # True
```

Iterator

- **Definition**: An object that represents a stream of data, returning one element at a time **when `next()` is called**.
- Created by calling `iter()` on an iterable.

- Must implement **two methods**:
    - `__iter__()` — returns the iterator object itself.
    - `__next__()` — returns the next value, or raises `StopIteration` when no more items.
- **One-time use**: Once exhausted, you can't reuse it without creating a new one.

```
my_list = [1, 2, 3]
my_iter = iter(my_list)  # Create iterator from iterable

print(next(my_iter))  # 1
print(next(my_iter))  # 2
print(next(my_iter))  # 3
# print(next(my_iter))  # Raises StopIteration
```

Feature	Iterable	Iterator
What	Can return an iterator	Gives one element at a time
Method	`__iter__()`	`__iter__()` + `__next__()`
Examples	list, tuple, str, dict, set	object from `iter(iterable)`
Reusable	Yes	No (once exhausted)

What is a Generator?

- A **special type of iterator**.
- Created using **functions with `yield`** instead of `return`.
- Generates values **lazily** (one at a time, only when requested) → saves memory.

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count   # Instead of return
        count += 1

# Using the generator
numbers = count_up_to(5)
print(next(numbers))  # 1
print(next(numbers))  # 2
print(next(numbers))  # 3
# And so on...
```

Why use `yield`?

- `return` → exits function immediately.

- `yield` → pauses the function, remembers where it left off, and resumes on the next `next()` call.

****

How Generators Work

1. A generator function contains **one or more `yield` statements**.
2. When called, it **does not run immediately**; it returns a **generator object**.
3. Use:
    - `next(generator)` → to get the next value.
    - `for` loop → to iterate automatically until `StopIteration` is raised.

What is a Decorator?

- A **function that takes another function as input** and returns a modified version of it.
- Commonly used for **logging, authentication, performance monitoring, caching**, etc.
- Syntax: `@decorator_name`

Decorator Without Function Arguments

```
def simple_decorator(func):
    def wrapper():
        print("Before function runs")
        func()
        print("After function runs")
    return wrapper

@simple_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Decorator With Function Arguments

```
def decorator_with_args(func):
    def wrapper(*args, **kwargs):  # Accepts any arguments
        print(f"Function called with: args={args}, kwargs={kwargs}")
        return func(*args, **kwargs)
    return wrapper

@decorator_with_args
def add(a, b):
    return a + b

print(add(5, 3))
```

Real Use Case – Logging

```
import time

def log_execution(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        print(f"[LOG] Starting '{func.__name__}'...")
        result = func(*args, **kwargs)
        print(f"[LOG] Finished '{func.__name__}' in {time.time() - start_time:.4f} seconds")
        return result
    return wrapper

@log_execution
def slow_function():
    time.sleep(1)
    print("Doing something slow...")

slow_function()
```

A decorator **wraps** a function to add extra behavior without changing the function's code.

- Use `*args, **kwargs` in wrapper to handle any number of arguments.
- Can be applied to multiple functions with `@decorator_name`.
- Useful for **logging, authentication, performance tracking, caching**.

OOPS

Class and Object

- **Class** → Blueprint/template for creating objects.
- **Object** → An instance of a class (actual implementation of the blueprint

A **class** defines variables (attributes) and functions (methods).

- An **object** can access class attributes and methods using `.` notation.

Constructor in Python

- Constructor = special method `__init__()` that runs automatically when an object is created.
- Used to initialize object attributes.

What is `self`?

- `self` represents the **instance of the class**.
- It is the first parameter in **instance methods**, and it refers to **the specific object calling the method**.
- You can name it anything (technically), but `self` is a **convention**.
- Without `self`, you can't access **instance attributes** inside methods.

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand   # belongs to this object
        self.model = model

    def show_brand(self):
        print(f"This car brand is {self.brand}")

car1 = Car("Tesla", "Model S")
car2 = Car("Ford", "Mustang")

car1.show_brand()  # This car brand is Tesla
car2.show_brand()  # This car brand is Ford
```

`self.brand` in `car1.show_brand()` → refers to `car1`'s brand.

- `self.brand` in `car2.show_brand()` → refers to `car2`'s brand.

***

```
class Car:
    def honk(self, times):
        print(f"{self} honks {times} times")

car = Car()

car.honk(2)          # instance call → Python passes `self` automatically
Car.honk(car, 2)     # class call    → you pass `self` explicitly
```

INSTANCE VS STATIC VS CLASS METHODS

Instance Method

- **Default** type of method in a class.
- First parameter is always `self` — the instance calling the method.
- Can access **instance attributes** and **class attributes**.

Class Method

- Defined with `@classmethod` decorator.

- First parameter is `cls` — the class itself (not the instance).
- Can access **class attributes**, but not instance-specific data unless passed manually.

class Car:

wheels = 4

@classmethod

def change_wheels(cls, count): # class method

cls.wheels = count

print(f"Wheels changed to {cls.wheels}")

Car.change_wheels(6)   # Wheels changed to 6

c = Car()

c.change_wheels(8)   # Wheels changed to 8 (affects all cars!)

Static Method

- Defined with `@staticmethod` decorator.
- Does **not** get `self` or `cls` automatically.
- Works like a normal function but lives inside the class for logical grouping.

Feature	Instance Method	Class Method	Static Method
First arg	`self`	`cls`	none
Access instance data	✅	❌	❌
Access class data	✅	✅	❌
Needs `@decorator`	❌	✅	✅

`self.attr = value` → changes **only that object** (creates/overrides instance attribute).

- `ClassName.attr = value` → changes for **all instances** that do not have their own attribute with the same name

class Car:

wheels = 4 # class attribute

def __init__(self, brand):

```
        self.brand = brand

c1 = Car("Tesla")
c2 = Car("BMW")

print(c1.wheels, c2.wheels)  # 4 4  (comes from class attribute)

c1.wheels = 6  # sets instance attribute for c1 only

print(c1.wheels, c2.wheels)  # 6 4
print(Car.wheels)            # 4  (unchanged class attribute)
Car.wheels = 8
print(c1.wheels, c2.wheels)  # 6 8  (c1 still has its own value, c2 uses class attribute)
```

Inheritance in Python

Inheritance lets a class (**child / subclass**) get attributes and methods from another class (**parent / base class**).

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

class Child(Parent):
    pass

c = Child()
c.greet()  # Inherited from Parent
```

`super()` keyword

`super()` is used inside a subclass to call methods from its **parent class** without directly naming the parent.

Without Super:

```
class Vehicle:
    def __init__(self, wheels):
        self.wheels = wheels

class Car(Vehicle):
    def __init__(self, wheels, brand):
        Vehicle.__init__(self, wheels)  # directly calling parent
        self.brand = brand
```

With super:

```
class Vehicle:
    def __init__(self, wheels):
        self.wheels = wheels
        print(f"Vehicle with {self.wheels} wheels created")

class Car(Vehicle):
    def __init__(self, wheels, brand):
        super().__init__(wheels)  # calls parent __init__
        self.brand = brand
        print(f"Car brand: {self.brand}")

c = Car(4, "Tesla")
```

What is Encapsulation?

Encapsulation is the practice of **restricting direct access** to an object's internal data (attributes) and controlling it through **methods**.
It's like putting your valuables in a locker — you can't just grab them directly; you need the proper key.

In Python terms:

- Keep data (attributes) private/protected.

- Use **getter** and **setter** methods to access/modify them.

```
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder    # public
        self.__balance = balance                # private

    # Getter
    def get_balance(self):
        return self.__balance

    # Setter
    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance!")

# Usage
acc = BankAccount("Abhishek", 1000)

print(acc.get_balance())  # ✅ Access via getter

acc.set_balance(1500)     # ✅ Update via setter
print(acc.get_balance())

acc.set_balance(-500)     # ❌ Rejected

# Direct access won't work:
# print(acc.__balance)  # AttributeError
# But Python still allows name mangling access:
print(acc._BankAccount__balance)  # 1500 (not recommended)
```

Why Use Encapsulation?

- Protects data from accidental modification.
- Adds validation before changing values.
- Makes your class easier to maintain.

Abstraction

Definition:
Abstraction means **showing only the essential details** to the user and hiding the background complexity.
In Python, this is usually done using **abstract classes** and **abstract methods** from the `abc` module.

Analogy:
When you drive a car, you just turn the steering wheel or press the accelerator — you don't need to know *how* the engine burns fuel or how gears shift.

```
from abc import ABC, abstractmethod


class Vehicle(ABC):  # Abstract Base Class
    @abstractmethod
    def start_engine(self):
        pass  # Abstract method — no implementation


class Car(Vehicle):
    def start_engine(self):
        print("Car engine started with key!")


class Bike(Vehicle):
    def start_engine(self):
        print("Bike engine started with self-start button!")


# Usage
v1 = Car()
v1.start_engine()
```

```python
v2 = Bike()
v2.start_engine()

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

Key Points:

- Abstract methods must be implemented in child classes.
- Prevents direct creation of incomplete classes.
- Defines a **template** for derived classes.

Polymorphism

Definition:
Polymorphism means **one name, many forms** — the same method name can behave differently depending on the object that calls it.

Types in Python:

1. **Method Overriding** (runtime polymorphism)
2. **Method Overloading** (Python doesn't support true overloading, but can mimic using default arguments or `*args`).

```python
class Animal:
    def speak(self):
        print("Animal makes a sound")


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


class Cat(Animal):
    def speak(self):
        print("Cat meows")


# Polymorphic behavior
animals = [Dog(), Cat(), Animal()]
```

```
for a in animals:
    a.speak()  # Same method name, different behavior
```

Concept	Purpose
Encapsulation	Restrict access to data & control via methods
Abstraction	Hide implementation, show only interface
Polymorphism	Same method name, different behavior

String manipulation problems

- List/dict/set comprehension problems
- Using built-in functions like `map`, `filter`, `reduce`
- OOP-based coding tasks (design a class, implement inheritance)
- Generators & iterators in practice
- File read/write problems
- API calls with `requests` module
- Small automation-like tasks (CSV read, JSON parse)