Python topics roadmap

Can you share me all the possible topics in Python and also advanced topics except the ML, AI, and all other stuff?

Absolutely! Here is a **complete, structured roadmap of all important Python topics**, from basics to advanced, **excluding ML/AI/data science frameworks**. This covers everything needed for development, interviews, backend engineering, scripting, automation, and system-level programming.

🔥 Python Topics Roadmap (Beginner → Advanced)

---

1. Python Basics

- Introduction to Python, installation, IDEs
- Python syntax and indentation
- Variables & Data Types
- Type Casting
- Input & Output
- Comments
- Operators (Arithmetic, Logical, Comparison, Bitwise)

Example

```python
a, b = 10, 5
print(a + b, a > b, a & b)
```

---

2. Control Flow

- If / elif / else
- Loops: for, while
- Loop control statements: break, continue, pass
- List comprehensions

Example

```python
```

```
nums = [i*i for i in range(10) if i % 2 == 0]
```

---

3. Data Structures

- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Arrays (from array module)
- Stacks / Queues using list & deque
- PriorityQueue (heapq)

---

4. Functions

- Defining & calling functions
- Arguments: positional, keyword, default, variable-length (*args, **kwargs)
- Lambda functions
- Recursion
- Decorators (important)
- Closures

---

5. Modules & Packages

- Importing modules
- Standard library modules: `math`, `datetime`, `os`, `sys`, `random`, `statistics`, `json`, `re`
- Creating custom modules & packages
- Virtual environments

---

6. File Handling

- Reading & writing files (txt, json, csv)
- Working with binary files
- File pointers & `with` context manager

---

7. Exception & Error Handling

- Try / except / finally / else
- Custom exceptions
- Raising exceptions

Intermediate Topics

8. Object-Oriented Programming (OOP)

- Classes & Objects
- Constructor
- Encapsulation
- Inheritance
- Polymorphism
- Abstract classes & Interfaces
- Method Overloading / Overriding
- `super()` and MRO
- Dataclasses

---

9. Iterators & Generators

- `__iter__()` and `__next__()`
- `yield` keyword
- Generator expressions

---

10. Regular Expressions (RegEx)

- Pattern matching
- Validation
- Searching and replacing

---

11. Python Memory Management

- Stack vs Heap
- Garbage Collector
- Reference counting
- `id()` and `del()`

12. Comprehensions

- List, Set, Dict & Generator comprehensions

---

13. Python Standard Libraries (Important)

- `os`, `sys`, `shutil`
- `subprocess`
- `threading`, `multiprocessing`
- `functools`
- `itertools`
- `collections`

Advanced Python Topics (Non-AI)

14. Functional Programming

- `map()`, `filter()`, `reduce()`
- Pure functions
- Immutability

---

15. Concurrency & Parallelism

- Threading
- Multiprocessing
- Asyncio (async/await)
- GIL (Global Interpreter Lock)
- Futures & Executors

---

16. Network & Socket Programming

- Sockets TCP/UDP
- HTTP basics
- Building custom protocols

---

17. API, Automation & Scripting

- REST APIs with `requests`
- Automate system tasks
- Web scraping using `BeautifulSoup`, `Selenium`

---

18. Database Handling

- SQLite
- MySQL / PostgreSQL connectors
- CRUD operations
- ORM Basics (SQLAlchemy)

---

19. Logging & Debugging

- `logging` library
- Debugging with `pdb`

---

20. Packaging & Deployment

- `pip`, `setup.py`, `pyproject.toml`
- Publishing package to PyPI
- Virtual environments (venv, pipenv, poetry)

---

21. Advanced OOP Patterns

- Design patterns (Singleton, Factory, Observer, Strategy)
- SOLID principles
- Clean architecture

---

22. Testing

- Unit testing: `unittest`, `pytest`
- Mocking
- Coverage

---

23. Web Development

- FastAPI / Flask fundamentals
- Routing
- Middleware
- Templating
- Authentication

---

24. GUI Development

- Tkinter
- PyQt
- Kivy

---

25. Cybersecurity & Ethical Hacking (Python Use)

- Cryptography basics
- Hashing
- Network scanning (scapy)

---

26. System & IoT Programming

- Interaction with hardware (Raspberry Pi)
- Serial communication

Bonus Useful Topics

- YAML, JSON, XML parsing
- Multithreaded downloads
- Event-driven programming
- Image processing basics (Pillow, OpenCV without ML)

🚀 What you can build after mastering

Project	Related Topics
Chat application	sockets, threading, json
Automation scripts	os, subprocess, requests

Project	Related Topics
Web app backend	Flask / FastAPI, DB, auth
File encryptor	cryptography, os
Terminal-based tools	argparse, sys
Package for pip	packaging concepts

Next Step

If you want, I can share:

- A week-by-week learning plan
- Project list based on each topic
- Advanced interview questions for Python

Would you like:

A → Full Python Roadmap with Projects

or

B → Python Backend Development Roadmap (Flask/FastAPI)?

Just tell **A or B** 😊

Now start explaining me everything, one by one, topic.

Perfect — we will start **from the very beginning** and move step-by-step through every topic in depth.
Today's topic is **Python Basics** (Lesson 1).
After completing this, we will move to **Control Flow**, **Data Structures**, **Functions**, and so on.

📍 Lesson 1 — Python Basics

1. What is Python?

Python is a high-level, interpreted, general-purpose programming language.
Known for:

- Simple and readable syntax
- Huge standard library
- Cross-platform support
- Works for app development, automation, backend, scripting, cybersecurity, etc.

---

2. How Python Works

- You write code in `.py` files.
- Python interpreter executes code line by line.
- No need to compile manually.

---

3. Python Syntax & Indentation

Python uses **indentation instead of curly braces** `{}` to define blocks.

Example

```python
x = 10
if x > 5:
    print("Greater")
```

Wrong code (missing indentation):

```python
if x > 5:
print("Greater")  # Error
```

Indentation default size: **4 spaces**.

---

4. Variables in Python

- Store values in memory
- Created automatically when assigned (no need to declare type)

Example

```python
name = "Priyanshu"
age = 21
pi = 3.14
is_active = True
```

---

5. Data Types

Type	Example
`int`	10
`float`	10.5
`str`	"Hello"
`bool`	True / False
`list`	[1, 2, 3]
`tuple`	(1, 2, 3)
`set`	{1, 2, 3}
`dict`	{ "name": "Sam", "age": 20 }
`NoneType`	None

---

6. Type Casting

Convert one data type to another.

```python
a = "10"
b = int(a)   # convert string to integer
print(type(b))
```

---

7. Input & Output

Output

```python
print("Hello, world!")
```

Input

```python
name = input("Enter your name: ")
print("Welcome", name)
```

Input always returns `str`

Convert if needed:

```python
age = int(input("Enter age: "))
```

---

8. Operators

Arithmetic

```python
a = 10
b = 3
print(a + b, a - b, a * b, a / b, a // b, a % b, a ** b)
```

Assignment

```python
x = 5
x += 2   # x = 7
```

Comparison

```python
```

```python
print(5 > 3, 5 == 5, 5 != 4)
```

Logical

```python
print(True and False, True or False, not True)
```

Bitwise

```python
print(5 & 3, 5 | 3, 5 ^ 3)
```

---

9. Comments

```python
# Single line comment

"""
Multi
Line
Comment
"""
```

🎯 Practice Tasks for Lesson 1

Try these yourself:

Task 1

Take user input of name, age, and city, then print in a formatted sentence:

```vbnet
Input:
Name: Priyanshu
Age: 22
City: Ahmedabad
```

```
Output:
Hello Priyanshu, you are 22 years old from Ahmedabad.
```

Task 2

Perform all arithmetic operations on two numbers entered by the user.

Task 3

Write code to check whether a number is even or odd.

📌 Next Lesson

If you are ready, we will start **Lesson 2 → Control Flow (if/else, loops, break, continue, comprehension)** in the next message.

Tell me "Next" when you finish Lesson 1 practice 🌟

Take your time — I'm with you step by step.

Yes

Awesome — let's move ahead 🚀

📍 Lesson 2 — Control Flow in Python

Control flow allows your program to **make decisions** and **repeat actions**.

---

1. Conditional Statements (if / elif / else)

Used when you want the program to choose paths based on conditions.

Syntax

```python
if condition:
    # code block
elif another_condition:
    # code block
else:
    # code block
```

Example

```python
age = 18

if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible")
```

Multiple conditions

```python
marks = 85

if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 60:
    print("Grade C")
else:
    print("Failed")
```

---

2. Nested If

```python
num = 10

if num > 0:
    if num % 2 == 0:
        print("Positive Even")
```

---

3. Loops

Loops repeat a block of code multiple times.

FOR Loop

Use when you know the number of iterations.

```python
for i in range(5):
    print(i)
```

Output:

```
0 1 2 3 4
```

Looping through lists

```python
fruits = ["apple", "banana", "mango"]

for item in fruits:
    print(item)
```

WHILE Loop

Use when you don't know the number of iterations beforehand.

```python
i = 1
while i <= 5:
    print(i)
    i += 1
```

---

4. Loop Control Statements

break

Stops the loop immediately.

```python
```

```python
for i in range(10):
    if i == 5:
        break
    print(i)
```

continue

Skip the current iteration and continue.

```python
for i in range(6):
    if i == 3:
        continue
    print(i)
```

pass

Placeholder for future code.

```python
for i in range(5):
    pass
```

---

5. List Comprehension

Short way to create lists.

Example

```python
squares = [x*x for x in range(6)]
print(squares)
```

Output

```csharp
[0, 1, 4, 9, 16, 25]
```

With condition

```python
even_nums = [i for i in range(10) if i % 2 == 0]
```

🎯 Practice Tasks for Lesson 2

Task 1

Ask user for a number and:

- Print whether it's positive, negative or zero.

Task 2

Print multiplication table of any number using a loop.

Task 3

Print the sum of numbers from 1 to 100 using a loop.

Task 4

Create a list of even numbers from 1 to 50 using list comprehension.

Task 5

Write a program to count number of vowels in a string.

📌 Next Lesson

When you're done practicing, tell me **"Next"** and we will begin:

Lesson 3 → Data Structures (Strings, Lists, Tuples, Sets, Dictionaries, Stack/Queue)

Take your time — mastering basics strongly is the key to becoming unbeatable 💪⚡

Yes

Amazing — let's continue with the next step 🚀

📍 Lesson 3 — Python Data Structures

Python has powerful built-in data structures, extremely important for development and DSA.

Today we will cover:

1. Strings
2. Lists
3. Tuples
4. Sets
5. Dictionaries
6. Stack & Queue basics

---

1. Strings

Strings are sequences of characters inside quotes.

Creating strings

```python
name = "Priyanshu"
```

Common string operations

```python
text = "hello world"
print(text.upper())      # HELLO WORLD
print(text.title())      # Hello World
print(text.replace("hello", "hi"))
print(len(text))
```

Indexing & Slicing

```python
s = "Python"
print(s[0])      # P
print(s[-1])     # n (last char)
print(s[1:4])    # yth (substring)
print(s[:3])     # Pyt
print(s[3:])     # hon
```

String iteration

```python
for char in "python":
    print(char)
```

---

2. Lists

Lists are **mutable**, ordered collections.

```python
items = [10, 20, 30, "apple"]
```

List operations

```python
items.append(40)        # add at end
items.insert(1, 15)     # add at index
items.remove(20)        # remove value
items.pop()             # remove last
items.sort()            # sort ascending
```

Accessing elements

```python
print(items[0])        # first
print(items[-1])       # last
```

Loop through list

```python
for i in items:
    print(i)
```

---

3. Tuples

Tuples are **immutable**, ordered lists.

```python
t = (10, 20, 30, "apple")
```

Example usage

```python
print(t[0])
```

Cannot modify tuple

```python
t[1] = 50   # Error
```

Useful for **fixed data**, faster than lists, used in databases, coordinates, etc.

---

4. Sets

Unordered collection of unique values.

```python
s = {1, 2, 3, 4, 4}   # duplicates removed
print(s)              # {1, 2, 3, 4}
```

Set operations

```python
s.add(5)
s.remove(2)
```

Set math operations

```python
a = {1, 2, 3}
b = {3, 4, 5}
```

```python
print(a | b)  # union -> {1, 2, 3, 4, 5}
print(a & b)  # intersection -> {3}
print(a - b)  # difference -> {1, 2}
```

---

5. Dictionaries

Key-value pair collection.

```python
student = {
    "name": "Priyanshu",
    "age": 21,
    "city": "Ahmedabad"
}
```

Access values

```python
print(student["name"])
print(student.get("age"))
```

Add/Update

```python
student["grade"] = "A"
```

Loop through dictionary

```python
for key, value in student.items():
    print(key, value)
```

---

6. Stack & Queue Using List/Deque

Stack (LIFO)

```python
```

```
stack = []
stack.append(10)
stack.append(20)
stack.append(30)
stack.pop()   # removes 30
```

Queue (FIFO)

```python
from collections import deque

queue = deque()
queue.append(10)
queue.append(20)
queue.append(30)
queue.popleft()   # removes 10
```

🎯 Practice Tasks for Lesson 3

Task 1

Reverse a string using slicing.

Task 2

Remove duplicates from a list using a set.

Task 3

Count frequency of each character in a string using a dictionary.

Task 4

Merge two lists element-wise using a loop or comprehension.

Task 5

Simulate stack push/pop operations and print final content.

📌 Next Lesson

Once done, say **"Next"** and we will start:

Lesson 4 — Functions & Lambda, Args, Kwargs, Recursion, Decorators

These are extremely important for interviews and backend development.

Say **Next** when ready 🌟

Yes

Excellent — moving forward confidently 🚀

📍 Lesson 4 — Functions in Python

Functions allow you to **reuse code**, **organize logic**, and **make programs cleaner & scalable**.

---

1. Defining & Calling Functions

Syntax

```python
def function_name():
    # code block
```

Example

```python
def greet():
    print("Hello, welcome!")

greet()
```

---

2. Function with Parameters

```python
def add(a, b):
    print(a + b)
```

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

---

3. Function with Return Value

```python
def multiply(a, b):
    return a * b

result = multiply(4, 6)
print(result)
```

---

4. Default Arguments

```python
def welcome(name="Guest"):
    print("Hello", name)

welcome()
welcome("Priyanshu")
```

---

5. Keyword Arguments

```python
def intro(name, age):
    print(name, age)

intro(age=22, name="Priyanshu")
```

---

6. Variable-Length Arguments

`*args` → handles multiple positional arguments

```python
```

```python
def total(*numbers):
    print(sum(numbers))

total(1, 2, 3, 4)
```

`**kwargs` → handles multiple keyword arguments

```python
def details(**info):
    print(info)

details(name="Priyanshu", role="Engineer", city="Ahmedabad")
```

---

7. Lambda Functions (Anonymous Functions)

Short one-line functions.

```python
square = lambda x: x*x
print(square(5))
```

Another example:

```python
add = lambda a, b: a + b
print(add(2, 3))
```

---

8. Recursion

Function calling itself.

```python
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

```
print(fact(5))
```

Note: Base condition is necessary, else infinite recursion will occur.

---

9. Decorators (Advanced but Important)

Decorators modify the behavior of a function **without changing its code**.

Example

python

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

@outer
def hello():
    print("Hello world!")

hello()
```

Output

pgsql

```
Before function runs
Hello world!
After function runs
```

Used in **Flask, FastAPI, Logging, Authorization, Timing functions**, etc.

---

10. Docstrings

Explain the purpose of function.

python

```python
def add(a, b):
    """This function adds two numbers"""
    return a + b

print(add.__doc__)
```

🎯 Practice Tasks for Lesson 4

Task 1

Write a function to find the largest number among three numbers.

Task 2

Write a function to check if a string is a palindrome.

Task 3

Write a recursive function to print Fibonacci series up to N.

Task 4

Create a decorator that prints execution time of a function.

Task 5

Write a function that accepts variable arguments using `*args` and returns the maximum number.

📌 Next Lesson

Say "**Next**" to continue with:

Lesson 5 — Modules & Packages, Standard Library, Importing, Virtual Environments

This is crucial for real projects and backend development.

Tell me **Next** when you're ready 💪✨

Continue

Perfect — let's continue with the next important chapter 🚀

📍 Lesson 5 — Modules & Packages in Python

Modules and packages allow you to **organize large programs**, **reuse code**, and **import built-in or external libraries**.

---

1. What is a Module?

A **module** is simply a Python file (`.py`) that contains variables, functions, classes, etc.

Example: Creating a module

Create a file named `math_utils.py`

```python
def add(a, b):
    return a + b

def square(x):
    return x * x
```

Importing module

```python
import math_utils

print(math_utils.add(5, 3))
print(math_utils.square(4))
```

Importing specific members

```python
from math_utils import add

print(add(10, 5))
```

---

2. Built-in Modules

Python provides many modules already in the standard library.

Module	Usage
`math`	Mathematical functions
`random`	Random number generation
`datetime`	Date & time operations
`os`	File & directory operations
`sys`	Python interpreter control
`json`	Read/Write JSON data
`re`	Regular expressions
`time`	Delay & timing

Example using built-in modules

python

```python
import math
print(math.sqrt(16))
```

python

```python
import random
print(random.randint(1, 10))
```

---

3. What is a Package?

A **package** is a directory that contains multiple modules with an `__init__.py` file inside.

Example structure

markdown

```
my_package/
    __init__.py
```

```
    math_utils.py
    string_utils.py
```

Importing from a package

```python
from my_package.math_utils import add
```

---

4. Installing External Packages

Python uses **pip** to install third-party libraries.

Example:

```bash
pip install requests
```

Usage:

```python
import requests

response = requests.get("https://api.github.com")
print(response.status_code)
```

---

5. Virtual Environments

Helps to manage separate dependencies for each project.

Creating a virtual environment

```bash
python -m venv myenv
```

Activate venv

Windows:

```bash
myenv\Scripts\activate
```

macOS / Linux:

```bash
source myenv/bin/activate
```

Deactivate

```bash
deactivate
```

Install packages inside venv

```bash
pip install flask
```

---

6. requirements.txt

File containing all installed dependencies.

Generate automatically

```bash
pip freeze > requirements.txt
```

Install from requirements

```bash
pip install -r requirements.txt
```

🎯 Practice Tasks for Lesson 5

Task 1

Create a custom module containing 3 math functions and import them into another file.

Task 2

Use `random` module to generate 6 random OTP digits.

Task 3

Write code using `datetime` to print current date & time.

Task 4

Make a virtual environment and install `requests`, then call an API.

📌 Next Lesson

Say **"Next"** when ready, and we will continue with:

Lesson 6 — File Handling (read, write, open modes, CSV, JSON)

This is essential for backend, automation, and real-world projects.

Tell me **Next** and we move ahead 💪🔥

Yes

Great — let's keep the momentum strong 🚀

📍 Lesson 6 — File Handling in Python

File handling allows Python programs to **create, read, update, and delete files**.
Important for automation, backend development, logging, data processing, etc.

---

1. Opening Files

Syntax

```python
open(filename, mode)
```

Mode	Meaning	Description
`r`	Read	File must exist
`w`	Write	Overwrites file / creates new
`a`	Append	Adds new content without deleting existing
`r+`	Read + Write	File must exist
`w+`	Write + Read	Creates file
`a+`	Append + Read	Creates file if not exists

---

2. Reading Files

Read full content

```python
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

Read line by line

```python
file = open("data.txt", "r")
for line in file:
    print(line)
file.close()
```

Read specific number of characters

```python
file = open("data.txt", "r")
print(file.read(10))
file.close()
```

3. Writing to Files

```python
file = open("data.txt", "w")
file.write("Hello, this is a test file.")
file.close()
```

4. Appending to Files

```python
file = open("data.txt", "a")
file.write("\nThis is a new line.")
file.close()
```

5. Using `with` Statement (Recommended)

Automatically closes the file.

```python
with open("sample.txt", "r") as f:
    print(f.read())
```

6. File Methods

```python
f.read()        # read entire content
f.readline()    # read single line
f.readlines()   # returns list of lines
f.write()       # write content
f.seek()        # move pointer
f.tell()        # current position
```

7. Working with JSON Files

```python
import json

person = {"name": "Priyanshu", "age": 21}

with open("data.json", "w") as f:
    json.dump(person, f)     # write json
```

Read JSON

```python
with open("data.json", "r") as f:
    data = json.load(f)
    print(data)
```

---

8. Working with CSV Files

```python
import csv

with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Priyanshu", 21])
```

Read CSV

```python
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

---

9. Delete Files

```python
import os
os.remove("data.txt")
```

🎯 Practice Tasks for Lesson 6

Task 1

Create a file and write 5 lines about yourself.

Task 2

Write a program to count number of words in a text file.

Task 3

Create a CSV file of 5 students with name, roll, marks and read it.

Task 4

Save Python dict to JSON and load it back.

📌 Next Lesson

Say **Next** to continue with:

Lesson 7 — Exception Handling (try, except, finally, raise, custom exceptions)

Critical for building strong backend and bug-free applications.

Tell me **Next** when ready ✨

Yes

Amazing — let's move ahead with the next essential chapter 🚀

📍 Lesson 7 — Exception Handling in Python

Exceptions help us **handle errors gracefully** instead of crashing the program.
Used heavily in **backend development, APIs, file handling, database work, and production systems**.

---

1. What is an Exception?

An exception is an event that interrupts normal program execution.

Example of error without handling:

```python
a = 10 / 0   # ZeroDivisionError
```

---

2. `try` - `except` Block

```python
try:
    a = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

---

3. Multiple Except Blocks

```python
try:
    num = int("hello")
except ValueError:
    print("Invalid number")
except TypeError:
    print("Type error occurred")
```

---

4. Catching All Exceptions

```python
```

```python
try:
    x = 10 / 0
except Exception as e:
    print("Error:", e)
```

---

5. `else` Block

Runs only when no exception occurs.

```python
try:
    n = int(input("Enter number: "))
except ValueError:
    print("Invalid input")
else:
    print("Square:", n*n)
```

---

6. `finally` Block

Always executes (cleanup, close file, close DB connection).

```python
try:
    f = open("file.txt")
except FileNotFoundError:
    print("File missing")
finally:
    print("Execution finished")
```

---

7. Raising Custom Errors

```python
def check_age(age):
    if age < 18:
        raise ValueError("Age must be above 18")
    return "Allowed"
```

```
print(check_age(16))
```

---

8. Custom Exception Class

```python
class LoginError(Exception):
    pass

def login(username):
    if username != "admin":
        raise LoginError("Invalid username")

try:
    login("user")
except LoginError as e:
    print(e)
```

---

9. Real-Life Example: File Handling

```python
try:
    with open("data.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("File not found!")
```

🎯 Practice Tasks for Lesson 7

Task 1

Write a program that handles division by zero.

Task 2

Ask user for a number; handle wrong input using try-except.

Task 3

Create a custom exception for invalid password (< 6 characters).

Task 4

Write a function that raises an exception if marks < 40.

📌 Next Lesson

Say **Next** to continue with:

Lesson 8 — Object-Oriented Programming (OOP)

Classes, Objects, Constructors, Inheritance, Encapsulation, Polymorphism, Abstraction, Dunder methods

This is one of the most important topics in Python and interviews 💯

Ready? Say **Next** ✨

Continue

Great — continuing Exception Handling in Python with deeper insights and then we'll move to **Object-Oriented Programming (OOP)** after this segment.

🔹 6. Exception Handling in Python (Continued)

6.5 `finally` Block

- Runs **every time**, whether an exception occurs or not.
- Useful for cleanup tasks like closing files, database connections, or releasing resources.

Example

```python
try:
    file = open("data.txt", "r")
    print(file.read())
except FileNotFoundError:
```

```
    print("File not found.")
finally:
    print("Closing file resource.")
```

📌 Even if the file doesn't exist, **finally block still executes**.

6.6 Raising Exceptions Manually — `raise`

If you want to stop execution intentionally and throw a custom error:

```python
age = -1
if age < 0:
    raise ValueError("Age cannot be negative.")
```

6.7 Creating Custom Exceptions

You can define your own type of exception for specific error cases.

Example:

```python
class NegativeBalanceError(Exception):
    pass

balance = -500
if balance < 0:
    raise NegativeBalanceError("Balance cannot be negative")
```

6.8 Multiple Exceptions in One Block

```python
try:
    num = int(input("Enter number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError):
    print("Invalid input or division by zero occurred.")
```

6.9 Using `else` with try-except

- Runs only if **no exception occurs**.

```python
try:
    x = 5 / 1
except ZeroDivisionError:
    print("Cannot divide")
else:
    print("Successful execution")
```

🔥 Real-Life Use Case

Online Payment System

- Try block → processing transaction
- Except → handle errors like insufficient balance
- Else → confirm success
- Finally → close payment gateway session

```python
try:
    pay(amount)
except PaymentError:
    print("Payment Failed")
else:
    print("Transaction Successful")
finally:
    print("Secure session closed")
```

📌 Summary of Exception Handling

Keyword	Purpose
`try`	Code that might cause an error
`except`	Handles exception

Keyword	Purpose
`else`	Runs if no exception
`finally`	Runs always
`raise`	Throw custom exception
Custom Exception	Create own exception types

Next Topic

🔶 7. Object-Oriented Programming (OOP) in Python

We will cover:

1. **Class & Object**
2. **Constructors**
3. **Attributes & Methods**
4. **Types of Methods (Instance / Class / Static)**
5. **Encapsulation**
6. **Inheritance**
7. **Polymorphism**
8. **Abstraction**
9. **Magic / Dunder Methods**
10. **OOP Real Project Example**

❓ Ready to start with OOP (Object-Oriented Programming)?

If yes, we'll begin with:

👉 **What is OOP and why do we use it?**
👉 **How to define a class and object in Python**

Say "Yes" to continue. 🚀

Yes

Awesome — let's begin **Object-Oriented Programming (OOP) in Python** step-by-step with clarity and real examples.

🔶 7. Object-Oriented Programming (OOP) in Python

7.1 What is OOP?

OOP is a programming methodology that organizes code into **objects** — real-world entities having:

- **Attributes (data)**
- **Methods (functions/actions)**

Why do we use OOP?

Feature	Benefit
Reusability	Use existing code again (less repetition)
Scalability	Easy to expand features
Security	Encapsulation hides data
Maintainability	Code is structured and manageable
Real-world mapping	Works similar to real objects

Real Example

Think of a **Car**:

Property	Behavior
color, brand, model	start, brake, speedUp

So in code, **Car is a class**, and **Toyota, BMW, Tesla cars are objects**.

7.2 Class and Object Basics

Class

A blueprint for creating objects.

Object

An instance created from a class.

Example

```python
class Car:
    brand = "BMW"
    model = "X5"

# Creating object of class
car1 = Car()

print(car1.brand)
print(car1.model)
```

Output

```nginx
BMW
X5
```

7.3 Defining a Class with Constructor

A constructor is a special method: `__init__()`

Called automatically when object is created.

Example

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

car1 = Car("Tesla", "Model S")
car2 = Car("BMW", "i8")
```

```
print(car1.brand, car1.model)
print(car2.brand, car2.model)
```

Output

```
rust

Tesla Model S
BMW i8
```

7.4 Instance Methods

Methods defined inside a class that operate on object data.

```
python

class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def full_name(self):
        return f"{self.brand} {self.model}"

car1 = Car("Audi", "A6")
print(car1.full_name())
```

7.5 Types of Methods

Method Type	Keyword Used	Purpose
Instance Method	`self`	Access/modify instance data
Class Method	`@classmethod`, `cls`	Works with class variables
Static Method	`@staticmethod`	Utility function (no self/cls)

Example: All Method Types

```python
class Student:
    school = "ABC School"   # class variable

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

    def info(self):  # Instance method
        return f"{self.name} - {self.marks}"

    @classmethod
    def change_school(cls, new_school):
        cls.school = new_school

    @staticmethod
    def greet():
        print("Welcome Student!")

s1 = Student("Priyanshu", 90)
print(s1.info())

Student.change_school("XYZ School")  # Calling class method
Student.greet()                      # Calling static method
```

📌 Summary Till Now

- ✔ What is OOP
- ✔ Why OOP is needed
- ✔ Class & Object
- ✔ Constructor `__init__()`
- ✔ Instance Variables & Methods
- ✔ `@classmethod` and `@staticmethod`

Next Topics

➡ SOON Coming next:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**
5. **Magic (Dunder) Methods**
6. **Mini Project using OOP**

Ready for Encapsulation?

Say "**Yes**" to continue 🚀

Yes

Perfect — let's continue with **Encapsulation** in Python OOP.

🔐 7.6 Encapsulation in Python

📌 What is Encapsulation?

Encapsulation is the process of **protecting data** inside a class by **restricting direct access** from outside the class.

It allows controlled access using **getter** and **setter** methods.

Why Encapsulation?

Benefit	Explanation
Security	Sensitive data stays protected
Control	Validate or restrict values before changing
Abstraction	User interacts through methods, not internal structure

---

🧠 Public, Protected, and Private Members

Access Type	Syntax	Meaning
Public	`var`	Accessible everywhere
Protected	`_var`	Accessible within class & subclasses
Private	`__var`	Accessible only inside class

🔹 Public Example

python

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

p1 = Person("Priyanshu")
print(p1.name)   # Accessible directly
```

🔸 Protected Example

python

```python
class Person:
    def __init__(self, name):
        self._name = name   # protected variable

p = Person("Rahul")
print(p._name)     # Allowed but not recommended
```

🔒 Private Example

python

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

```
acct = BankAccount(5000)
print(acct.__balance)   # ❌ Error: cannot access private variable
```

Getter and Setter Methods

Access and modify private variables safely.

Example

python

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

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

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

acct = BankAccount(5000)
print(acct.get_balance())

acct.set_balance(8000)
print(acct.get_balance())

acct.set_balance(-2000)  # invalid update
```

Output

yaml

```yaml
5000
8000
Invalid amount!
```

🔥 Real-Life Example

ATM System

- Balance cannot be accessed directly.
- Only operations are:
  - Withdraw
  - Deposit
  - Check balance
- Balance must be safe from direct changes.

🗒️ Summary

- ✔️ Data hiding using private variables
- ✔️ `_protected` vs `__private`
- ✔️ Getter & Setter methods
- ✔️ Real-life ATM example

Next Topic

🌳 7.7 Inheritance

We will cover:

- What is inheritance?
- Types of inheritance
- `super()` keyword
- Real-life example

Say "Yes" to begin with Inheritance 🚀

Yew

Great! Let's dive into **Inheritance** in Python.

🌳 7.7 Inheritance in Python

📌 What is Inheritance?

Inheritance allows one class (**child**) to derive or inherit properties and methods from another class (**parent**).

Why use Inheritance?

Advantage	Explanation
Reusability	Use features of existing class
Extensibility	Add new features easily
Maintainability	Less repetitive code
Hierarchy modeling	Represents real relationships

🧠 Basic Example

```python
class Parent:
    def show(self):
        print("This is Parent class")

class Child(Parent):
    pass

c = Child()
c.show()
```

Output

```csharp
This is Parent class
```

👉 `Child()` inherited the `show()` method from `Parent`.

`__init__()` with Inheritance

Example

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

class Student(Person):
    def __init__(self, name, grade):
        super().__init__(name)   # call parent constructor
        self.grade = grade

s1 = Student("Priyanshu", "A")
print(s1.name, s1.grade)
```

Output

```css
Priyanshu A
```

👉 `super()` calls the parent's constructor.

💡 Types of Inheritance

Type	Example Structure
Single Inheritance	1 parent → 1 child
Multiple Inheritance	Child inherits from multiple parents
Multilevel Inheritance	Parent → Child → Grandchild
Hierarchical Inheritance	One parent → multiple children
Hybrid Inheritance	Combination of above types

Single Inheritance Example

python

```python
class Animal:
    def eat(self):
        print("Eating")

class Dog(Animal):
    def bark(self):
        print("Barking")

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

Multilevel Inheritance

python

```python
class A:
    def feature1(self):
        print("Feature 1")

class B(A):
    def feature2(self):
        print("Feature 2")

class C(B):
    def feature3(self):
        print("Feature 3")

c = C()
c.feature1()
c.feature2()
c.feature3()
```

Multiple Inheritance

python

```python
class Father:
    def sports(self):
        print("Father loves cricket")

class Mother:
    def music(self):
        print("Mother loves singing")

class Child(Father, Mother):
    pass

c = Child()
c.sports()
c.music()
```

🧠 Method Overriding

Child class can replace parent method.

```python
class Parent:
    def work(self):
        print("Parent works in office")

class Child(Parent):
    def work(self):
        print("Child studies at college")

obj = Child()
obj.work()
```

Output:

```nginx
Child studies at college
```

🔥 Real-Life Use Case Example

Animal Classification System

```python
class Animal:
    def sound(self):
        print("General animal sound")

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

class Cat(Animal):
    def sound(self):
        print("Meow")

animals = [Dog(), Cat()]
for a in animals:
    a.sound()
```

📌 Summary

- ✔ Inheritance allows code reuse
- ✔ `super()` calls parent constructor
- ✔ Supports multiple inheritance
- ✔ Method overriding changes parent behavior
- ✔ Real-life example: animals system

Next Topic

🌀 7.8 Polymorphism

Will cover:

- What is polymorphism?
- Method overloading vs overriding
- Built-in polymorphism
- Duck typing in Python

Say "Yes" to continue with Polymorphism 🚀

Yes

Excellent — let's continue with **Polymorphism** in Python.

🌀 7.8 Polymorphism in Python

📌 What is Polymorphism?

Polymorphism means **many forms** — the same function or method name behaves differently depending on the object.

Real-world example

- `+` works differently for numbers and strings
  - `10 + 20 = 30`
  - `"Hello" + "World" = HelloWorld"`
- A `Dog` and `Cat` both have `sound()` method but different outputs.

🔷 Polymorphism with Method Overriding

This happens when a **child class** changes the behavior of a method from the **parent class**.

python

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

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

class Cat(Animal):
    def sound(self):
        print("Meow")

animals = [Dog(), Cat()]
```

```
for animal in animals:
    animal.sound()
```

Output

```
nginx

Bark
Meow
```

👉 Same method name → different results.

🔷 Polymorphism with Built-in Functions

```
python

print(len("Priyanshu"))   # length of string
print(len([1, 2, 3, 4]))  # length of list
```

Output

```


9
4
```

👉 Same function `len()` works differently depending on input type.

🔷 Polymorphism with Classes

```
python

class Car:
    def fuel(self):
        print("Petrol or Diesel")

class ElectricCar:
    def fuel(self):
        print("Electric Charging")
```

```
for vehicle in (Car(), ElectricCar()):
    vehicle.fuel()
```

🦆 Duck Typing in Python

"If it walks like a duck and quacks like a duck, it's a duck."

Python doesn't care about the **type of object**; if it has the necessary **method**, it works.

Example

python

```
class Duck:
    def speak(self):
        print("Quack Quack")

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

def animal_sound(animal):
    animal.speak()

animal_sound(Duck())
animal_sound(Dog())
```

🔁 Method Overloading in Python

Python does **not** support true method overloading like Java/C++

But we simulate it using default arguments.

python

```
class Math:
    def add(self, a, b=0, c=0):
        return a + b + c

m = Math()
print(m.add(5))
```

```
print(m.add(5, 10))
print(m.add(5, 10, 15))
```

📌 Summary

Concept	Explanation
Polymorphism	Same function name, different behavior
Method Overriding	Child modifies parent method
Duck Typing	Works if behavior exists, not type
Overloading	Achieved using default arguments

Next Topic

🎭 7.9 Abstraction

We will learn:

- What is abstraction?
- Abstract classes
- `ABC` module & `@abstractmethod`
- Real-life example

Say “Yes” to continue with Abstraction 🚀

Yes

Great — let’s move into **Abstraction** in Python.

🎭 7.9 Abstraction in Python

📌 What is Abstraction?

Abstraction is the process of **hiding complex implementation details** and **showing only essential information** to the user.

Real-Life Example

- When you drive a car:
  - You press the accelerator → speed increases
  - You don't see internal engine mechanism (fuel pump, gearbox, ignition system)

Abstraction hides complexity and exposes only necessary controls.

📌 Abstraction in Python

In Python, abstraction is implemented using:

✔️ Abstract Classes

✔️ Abstract Methods

✔️ `ABC` Module (Abstract Base Class)

🔧 Abstract Class and Abstract Method

An **abstract class** contains one or more **abstract methods** (methods without implementation).

We create them using:

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

🔹 Example

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
```

```
    @abstractmethod
    def start(self):
        pass

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

class Bike(Vehicle):
    def start(self):
        print("Bike starts with self start button")

v1 = Car()
v2 = Bike()

v1.start()
v2.start()
```

Output

```
python

Car starts with key
Bike starts with self start button
```

👉 `Vehicle` class defines the common structure

👉 Actual implementation is done in child classes

🔒 Key Rules

Rule	Description
Abstract class cannot be instantiated	`obj = Vehicle()` ❌
Must override all abstract methods	Otherwise error
Can contain concrete (normal) methods	Not limited to abstract methods

---

🔥 Real Use Case

Payment System

Different payment types (UPI, Credit Card, Net Banking) behave differently but share a common structure.

```python
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def make_payment(self, amount):
        pass

class UPI(Payment):
    def make_payment(self, amount):
        print(f"Paid {amount} via UPI")

class Card(Payment):
    def make_payment(self, amount):
        print(f"Paid {amount} via Card")

pay1 = UPI()
pay1.make_payment(500)

pay2 = Card()
pay2.make_payment(1000)
```

📌 Summary of Abstraction

Concept	Meaning
Abstraction	Hiding internal logic, showing required interface
Abstract Class	Blueprint for other classes
Abstract Method	Must be implemented in subclass
ABC Module	Used to define abstract classes

Next Topic

⚙️ 7.10 Magic (Dunder) Methods

We will learn:

- What are magic methods?
- `__init__`, `__str__`, `__len__`, `__add__`, `__eq__`
- Operator overloading examples

Say "Yes" to continue with Magic/Dunder Methods 🚀

Yes

Awesome — let's continue with **Magic (Dunder) Methods** in Python.
These are extremely powerful and widely used in professional codebases.

⚙️ 7.10 Magic / Dunder Methods in Python

📌 What are Magic/Dunder Methods?

Magic methods (also called **Dunder** because they start and end with **double underscores** `__`) are special predefined methods in Python that allow **custom behavior** for built-in operations such as:

- printing objects
- comparing objects
- adding objects
- calling objects as functions
- indexing, length operations, etc.

Examples of Dunder Methods

Category	Method	Meaning
Constructor	`__init__`	initializes object
Representation	`__str__`, `__repr__`	convert object to string

Category	Method	Meaning
Operator overloading	`__add__`, `__sub__`, `__mul__`	operator behavior
Comparison	`__eq__`, `__lt__`, `__gt__`	comparison behavior
Container behavior	`__len__`, `__getitem__`, `__setitem__`	list-like behavior
Callable	`__call__`	object behaves like function

🔹 Example: `__str__()`

Used for readable output.

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

    def __str__(self):
        return f"{self.name} scored {self.marks}"

s1 = Student("Priyanshu", 95)
print(s1)
```

Output:

```nginx
Priyanshu scored 95
```

🔹 Operator Overloading Using `__add__()`

```python
```

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

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

p1 = Point(2, 3)
p2 = Point(4, 1)
p3 = p1 + p2
print(p3.x, p3.y)
```

Output:

```
6 4
```

👉 Now `+` works for objects like numbers.

🔹 Comparison Overloading: `__eq__()`

```
python

class Book:
    def __init__(self, pages):
        self.pages = pages

    def __eq__(self, other):
        return self.pages == other.pages

b1 = Book(200)
b2 = Book(200)
print(b1 == b2)
```

Output

```
graphql

True
```

🔹 `__len__()` Example

```python
class Course:
    def __init__(self, students):
        self.students = students

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

obj = Course(["A", "B", "C"])
print(len(obj))
```

Output

```
3
```

🔹 `__call__()` → Object as a Function

```python
class Printer:
    def __call__(self, text):
        print("Printing:", text)

p = Printer()
p("Hello World")   # behaves like function
```

📌 Summary

Feature	Explanation
Magic methods	Start/end with double underscores
Used to modify behavior	printing, comparison, arithmetic, calling

Feature	Explanation
Enables	cleaner, readable, custom object behavior

Next Topic

📦 7.11 Modules & Packages

We will cover:

- What is a module?
- What is a package?
- Importing modules
- Creating custom modules & packages
- `__init__.py`

Say “Yes” to continue with Modules & Packages 🚀

Yes

Great — let’s continue with **Modules & Packages** in Python.

📦 7.11 Modules & Packages in Python

📌 What is a Module?

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

Example

Create a file:

```python
# file: mymodule.py
def add(a, b):
```

```
    return a + b
```

Use it in another file:

```python
import mymodule

print(mymodule.add(5, 10))
```

💡 Why use Modules?

Benefit	Explanation
Reusability	Write once, use anywhere
Code Organization	Split large programs into separate files
Maintainability	Easy to update
Avoids Duplicate Code	Centralized logic

📌 Importing Modules

Method	Example
Import complete module	`import math`
Import specific function	`from math import sqrt`
Rename module	`import numpy as np`
Import all	`from math import *` (not recommended)

Example:

```python
from math import sqrt, pi
print(sqrt(16))
print(pi)
```

📦 What is a Package?

A **package** is a collection of modules stored in folders with an `__init__.py` file.

Folder Structure

```csharp
myproject/
   calculations/
        __init__.py
        add.py
        subtract.py
```

add.py

```python
def add(a, b):
    return a + b
```

subtract.py

```python
def sub(a, b):
    return a - b
```

Using Package

```python
from calculations.add import add
from calculations.subtract import sub

print(add(10, 5))
print(sub(10, 5))
```

🔹 Role of `__init__.py`

Feature	Purpose
Identifies folder as package	Enables import
Can perform initialization	pre-load config
Can control exposed imports	`__all__` usage

Example:

```python
__all__ = ["add", "sub"]
```

📌 Built-in Modules Examples

Module	Purpose
`math`	mathematical functions
`os`	operating system functions
`random`	random number generation
`datetime`	date/time
`sys`	system-level operations
`json`	JSON handling

Example: Random

```python
import random
```

```
print(random.randint(1, 10))
```

🔥 Real-Life Example: Utility Package

Useful in big projects like **Flask / Django / FastAPI**.

```
markdown

ecommerce/
   utils/
      __init__.py
      email.py
      notification.py
   payments/
      upi.py
      card.py
```

📌 Summary

Concept	Meaning
Module	Single Python file
Package	Group of modules in a folder
Import	Use code from another file
`__init__.py`	Marks directory as package

Next Topic

📂 7.12 File Handling

We will learn:

- Opening & closing files
- Read / Write / Append modes

- Working with `with open()`
- File operations: read(), write(), readline(), readlines()
- Handling exceptions during file processing

Say “Yes” to continue with File Handling 🚀

Yes

Perfect — let’s continue with **File Handling** in Python. This is essential for reading/writing data in real-world projects.

📂 7.12 File Handling in Python

📌 Opening a File

Use the built-in `open()` function:

```python
file = open("example.txt", "r")  # "r" = read mode
```

File Modes

Mode	Description
`r`	Read (file must exist)
`w`	Write (creates new or overwrites existing)
`a`	Append (adds at end)
`r+`	Read and write
`w+`	Write and read (overwrites)
`a+`	Append and read

📌 Reading Files

Read entire content

```python
with open("example.txt", "r") as f:
    data = f.read()
    print(data)
```

Read line by line

```python
with open("example.txt", "r") as f:
    for line in f:
        print(line.strip())  # remove newline
```

Read specific number of characters

```python
with open("example.txt", "r") as f:
    print(f.read(10))  # read first 10 chars
```

📌 Writing to Files

```python
with open("example.txt", "w") as f:
    f.write("Hello World!\n")
    f.write("Python File Handling")
```

- **Note:** `w` mode overwrites existing content.
- Use `a` mode to append without deleting existing content.

```python
with open("example.txt", "a") as f:
    f.write("\nAppended line")
```

📌 File Methods

Method	Purpose
`read()`	Reads entire file
`readline()`	Reads one line
`readlines()`	Reads all lines into a list
`write()`	Write text to file
`seek()`	Move cursor to a position
`tell()`	Current position of cursor

📌 Working with CSV Files

python

```python
import csv

# Writing CSV
with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "Age", "Grade"])
    writer.writerow(["Priyanshu", 21, "A"])

# Reading CSV
with open("students.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

📌 Working with JSON Files

python

```python
import json
```

```python
data = {"name": "Priyanshu", "age": 21}

# Writing JSON
with open("data.json", "w") as f:
    json.dump(data, f)

# Reading JSON
with open("data.json", "r") as f:
    loaded_data = json.load(f)
    print(loaded_data)
```

📌 Deleting Files

```python
import os

os.remove("example.txt")  # permanently deletes file
```

🔥 Real-Life Use Case

- Reading configuration files for applications
- Logging user activity
- Storing temporary data
- Exporting data to CSV / JSON

📌 Summary

- ✔ `open()` to open files
- ✔ Modes: `r`, `w`, `a`, `r+`, `w+`, `a+`
- ✔ `with` ensures file is closed automatically
- ✔ `csv` and `json` modules for structured data
- ✔ `os.remove()` deletes files

Next Topic

📊 7.13 Working with Dates and Times

- `datetime` module
- Current date & time
- Formatting dates
- Time arithmetic

Say “Yes” to continue with Dates & Times 🚀

Start from 9

Got it! We’ll start from **Lesson 9 — Dates & Times in Python**. This is crucial for logging, scheduling, and time-based operations.

🗓️ 9. Working with Dates and Times in Python

Python provides the `datetime` module for handling **date and time** objects.

```python
import datetime
```

9.1 Current Date and Time

```python
import datetime

now = datetime.datetime.now()
print(now)           # full date and time
print(now.date())    # only date
print(now.time())    # only time
```

Sample Output

```yaml
2025-12-06 08:30:15.123456
2025-12-06
```

```
08:30:15.123456
```

9.2 Creating Date or Time Objects

```python
# Specific date
d = datetime.date(2025, 12, 6)
print(d)

# Specific time
t = datetime.time(14, 30, 45)
print(t)
```

9.3 Accessing Date and Time Components

```python
now = datetime.datetime.now()

print(now.year)
print(now.month)
print(now.day)
print(now.hour)
print(now.minute)
print(now.second)
```

9.4 Formatting Dates and Times

Use `strftime()` to format and `strptime()` to parse strings.

```python
now = datetime.datetime.now()

# Format: DD/MM/YYYY
formatted = now.strftime("%d/%m/%Y")
print(formatted)
```

```python
# Format: Month name and time
formatted2 = now.strftime("%B %d, %Y %H:%M:%S")
print(formatted2)
```

9.5 Time Delta (Date Arithmetic)

```python
from datetime import datetime, timedelta

today = datetime.now()
tomorrow = today + timedelta(days=1)
yesterday = today - timedelta(days=1)

print("Today:", today)
print("Tomorrow:", tomorrow)
print("Yesterday:", yesterday)
```

Other operations:

- `timedelta(weeks=1)` → add/subtract a week
- `timedelta(hours=5)` → add/subtract hours

9.6 Comparing Dates

```python
d1 = datetime.date(2025, 12, 6)
d2 = datetime.date(2025, 12, 10)

if d1 < d2:
    print("d1 is before d2")
else:
    print("d1 is after d2")
```

9.7 Real-Life Use Cases

- Scheduling tasks or reminders
- Calculating age
- Logging events with timestamps

- Generating reports by date

📌 Summary

- `datetime` module for date & time
- `now()`, `date()`, `time()` for current values
- `strftime()` for formatting, `strptime()` for parsing
- `timedelta` for date arithmetic
- Compare dates using `<`, `>`, `==`

Next Lesson

10. Regular Expressions (Regex) in Python

- Matching patterns in text
- Validation (emails, phone numbers)
- Searching, replacing, splitting strings

Say "Yes" to continue with Regular Expressions 🚀

Yes

Perfect! Let's dive into **Lesson 10 — Regular Expressions (Regex) in Python**. Regex is extremely powerful for **pattern matching, searching, and validating text**.

🔍 10. Regular Expressions (Regex) in Python

Python provides the `re` module to work with **regular expressions**.

```python
import re
```

10.1 Basic Functions in `re`

Function	Description
`re.match()`	Check if pattern matches at the **start** of string
`re.search()`	Search pattern **anywhere** in string
`re.findall()`	Return **all matches** as a list
`re.split()`	Split string by pattern
`re.sub()`	Replace pattern with new string

10.2 Basic Example

```python
import re

text = "My email is priyanshu@example.com"

# Search for "email" in text
result = re.search("email", text)
if result:
    print("Found!")
else:
    print("Not found")
```

10.3 Using `re.findall()`

```python
text = "Call 123-456-7890 or 987-654-3210"

numbers = re.findall(r"\d{3}-\d{3}-\d{4}", text)
print(numbers)
```

Output

```css
```

```
['123-456-7890', '987-654-3210']
```

10.4 Validating Email

```python
pattern = r"[a-zA-Z0-9._]+@[a-zA-Z]+\.[a-zA-Z]{2,4}"
email = "priyanshu@example.com"

if re.match(pattern, email):
    print("Valid Email")
else:
    print("Invalid Email")
```

10.5 Replacing Text (`re.sub`)

```python
text = "Python is great"
new_text = re.sub(r"great", "awesome", text)
print(new_text)
```

Output

```csharp
Python is awesome
```

10.6 Splitting Strings with Regex

```python
text = "apple, banana; orange|grape"
fruits = re.split(r"[;,|]\s*", text)
print(fruits)
```

Output

```css
```

```
['apple', 'banana', 'orange', 'grape']
```

10.7 Common Regex Patterns

Pattern	Meaning
`.`	Any character except newline
`^`	Start of string
`$`	End of string
`*`	0 or more occurrences
`+`	1 or more occurrences
`?`	0 or 1 occurrence
`\d`	Digit `[0-9]`
`\D`	Non-digit
`\w`	Alphanumeric `[a-zA-Z0-9_]`
`\W`	Non-alphanumeric
`\s`	Whitespace
`\S`	Non-whitespace

🔥 Real-Life Use Cases

- Form validation (email, phone, password)
- Log file parsing
- Web scraping for specific patterns
- Data cleaning

📌 Summary

- `re` module is for regex operations
- Use `match`, `search`, `findall`, `sub`, `split`
- Patterns allow powerful string manipulation
- Widely used in automation, scraping, and data validation

Next Topic

11. Decorators in Python

- What are decorators?
- Function decorators
- Using `@decorator` syntax
- Parameterized decorators
- Real-life examples (logging, authentication)

Say "Yes" to continue with Decorators 🚀

Yes

Awesome! Let's move on to **Lesson 11 — Decorators in Python**. Decorators are widely used in **real-world Python projects** for adding functionality to functions or methods without modifying them.

🛠️ 11. Decorators in Python

11.1 What is a Decorator?

A **decorator** is a **function that takes another function and extends its behavior** without explicitly modifying it.

- Think of it as a **wrapper** around a function.

11.2 Basic Example

python

```python
def decorator(func):
    def wrapper():
        print("Before the function")
        func()
        print("After the function")
    return wrapper

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

# Using decorator manually
say_hello = decorator(say_hello)
say_hello()
```

Output

```pgsql
Before the function
Hello!
After the function
```

11.3 Using `@` Syntax

Python provides syntactic sugar to apply decorators easily:

```python
@decorator
def say_hello():
    print("Hello World!")

say_hello()
```

Output

```pgsql
Before the function
Hello World!
After the function
```

11.4 Decorator with Arguments

```python
def decorator(func):
    def wrapper(name):
        print(f"Hello, {name}!")
        func(name)
    return wrapper

@decorator
def greet(name):
    print("Have a great day!")

greet("Priyanshu")
```

Output

```css
Hello, Priyanshu!
Have a great day!
```

11.5 Decorator with Return Values

```python
def decorator(func):
    def wrapper(a, b):
        result = func(a, b)
        return result * 2
    return wrapper

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

print(add(5, 10))  # (5+10)*2 = 30
```

11.6 Chaining Decorators

```python
def decorator1(func):
    def wrapper():
        print("Decorator 1")
        func()
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2")
        func()
    return wrapper

@decorator1
@decorator2
def greet():
    print("Hello!")

greet()
```

Output

```nginx
Decorator 1
Decorator 2
Hello!
```

🔥 Real-Life Use Cases

- Logging function calls
- Timing execution
- Authorization / Authentication
- Caching results
- Retry logic for APIs

```python
import time

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

```
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Execution time: {end-start} seconds")
        return result
    return wrapper

@timer
def compute():
    sum(range(1000000))

compute()
```

📌 Summary

Concept	Explanation
Decorator	Function wrapping another function
`@decorator`	Applies decorator easily
With args/return	Works with functions having arguments and return values
Real Use Cases	Logging, auth, caching, timing

Next Topic

12. Generators in Python

- What are generators
- `yield` keyword
- Advantages over lists
- Real-life examples (large data handling)

Say “Yes” to continue with Generators 🚀

Yes

Perfect! Let's continue with **Lesson 12 — Generators in Python**, which are essential for **efficient memory management and handling large data**.

---

12. Generators in Python

12.1 What is a Generator?

A **generator** is a **special type of iterator** that **yields items one by one**, instead of storing all items in memory at once.

- Use `yield` instead of `return`
- Saves memory for large datasets

12.2 Basic Example

```python
def simple_gen():
    yield 1
    yield 2
    yield 3

gen = simple_gen()

print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

- Each `yield` pauses the function and remembers its state.
- When `next()` is called, it resumes from where it left off.

12.3 Iterating Over a Generator

```python
def numbers(n):
    for i in range(1, n+1):
        yield i
```

```
for num in numbers(5):
    print(num)
```

Output

```
1
2
3
4
5
```

12.4 Generator Expressions

Similar to list comprehensions but **lazy evaluation**.

```python
gen_exp = (x*x for x in range(5))

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

- Saves memory because it does **not create the entire list at once**

12.5 Advantages of Generators

Advantage	Explanation
Memory Efficient	Generates items on the fly
Lazy Evaluation	Only computes values when needed
Infinite Sequence	Can model infinite streams of data
Easy to Implement	Use `yield` keyword instead of building iterators manually

---

12.6 Real-Life Use Cases

- Reading large files line by line
- Streaming data from API or database
- Infinite sequences (like Fibonacci numbers)
- Pipelines for processing large data

Example: Reading large file

```python
def read_file_line_by_line(file_path):
    with open(file_path, 'r') as f:
        for line in f:
            yield line.strip()

for line in read_file_line_by_line("large_file.txt"):
    print(line)
```

📌 Summary

- Generators are **iterators with lazy evaluation**
- Use `yield` to produce values one by one
- Memory efficient for large datasets
- Can replace heavy lists for pipelines and streams

Next Topic

13. Context Managers and `with` Statement

- Why context managers exist
- `with` keyword for resource management
- Creating custom context managers
- Real-world examples (file handling, DB connections)

Say “Yes” to continue with Context Managers 🚀

Yes

Awesome! Let's continue with **Lesson 13 — Context Managers in Python**. This is key for **efficient resource management**.

🛡️ 13. Context Managers and `with` Statement

13.1 What is a Context Manager?

A **context manager** is a Python construct used to **manage resources** such as files, network connections, or database connections.

- Automatically handles **setup and cleanup**
- Ensures resources are **released properly** even if errors occur

---

13.2 Using `with` Statement

```python
with open("example.txt", "w") as f:
    f.write("Hello World!")
```

- `with` automatically closes the file after block execution
- No need to call `f.close()`

13.3 Why Use Context Managers?

Benefit	Explanation
Safety	Ensures resource is properly released
Cleaner Code	Reduces boilerplate `try/finally` blocks
Exception Handling	Closes resource even if error occurs

13.4 Custom Context Manager Using Class

```python
```

```python
class MyContext:
    def __enter__(self):
        print("Entering context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")

with MyContext() as mc:
    print("Inside the block")
```

Output

```scss
Entering context
Inside the block
Exiting context
```

- `__enter__()` → runs at the start of the block
- `__exit__()` → runs at the end of the block

13.5 Custom Context Manager Using `contextlib`

```python
from contextlib import contextmanager

@contextmanager
def my_context():
    print("Start")
    yield
    print("End")

with my_context():
    print("Inside")
```

Output

```powershell
Start
Inside
```

```
End
```

13.6 Real-Life Use Cases

- File handling (`open()`)
- Database connections (`sqlite3.connect`)
- Thread locks (`threading.Lock()`)
- Network connections (sockets)

Example: Database Connection

```python
import sqlite3

with sqlite3.connect("mydb.db") as conn:
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS users(name TEXT)")
```

- No need to manually close connection; context manager handles it

---

📌 Summary

- Context managers handle **setup and cleanup** automatically
- `with` is the Pythonic way to use them
- Can create custom context managers using **class** or **@contextmanager**
- Widely used in **file operations, DB connections, threads, and sockets**

Next Topic

14. Iterators in Python

- Difference between iterable and iterator
- `__iter__()` and `__next__()` methods
- Creating custom iterators
- Real-life examples (looping, generators)

Say "Yes" to continue with Iterators 🚀

Yes

Perfect! Let's continue with **Lesson 14 — Iterators in Python**. Iterators are fundamental for understanding **loops, generators, and custom iterable objects**.

🔁 14. Iterators in Python

14.1 What is an Iterator?

An **iterator** is an object that **represents a stream of data**.
It **produces items one at a time** when you call `next()` on it.

- **Iterable**: Can be looped over (like lists, tuples)
- **Iterator**: Produces values one by one using `__next__()`

14.2 Difference: Iterable vs Iterator

Feature	Iterable	Iterator
Can use `for` loop	✅	✅
Has `__iter__()`	✅	✅
Has `__next__()`	❌	✅
Produces next value	❌	✅

14.3 Getting an Iterator

python

```python
my_list = [1, 2, 3]
it = iter(my_list)   # create iterator

print(next(it))  # 1
print(next(it))  # 2
print(next(it))  # 3
# next(it)  # StopIteration if called again
```

- `iter()` → returns an iterator from iterable

- `next()` → gets the next element

14.4 Iterating Using a Loop

```python
my_list = [10, 20, 30]
for value in my_list:  # automatically uses iterator
    print(value)
```

14.5 Creating Custom Iterator

```python
class MyNumbers:
    def __init__(self, limit):
        self.num = 1
        self.limit = limit

    def __iter__(self):
        return self  # an iterator must return self

    def __next__(self):
        if self.num <= self.limit:
            val = self.num
            self.num += 1
            return val
        else:
            raise StopIteration

numbers = MyNumbers(5)
for n in numbers:
    print(n)
```

Output

```
1
2
3
```

```
4
5
```

14.6 Real-Life Use Cases

- Looping through large datasets
- Generating sequences on the fly
- Working with custom streams (file, API, data)
- Integrating with generators for memory efficiency

📌 Summary

- Iterators produce **one item at a time**
- Iterable → object that can return an iterator
- Iterator → object with `__iter__()` and `__next__()`
- StopIteration → signals end of iteration
- Custom iterators allow control over sequences

Next Topic

15. Exception Handling in Python

- What are exceptions
- `try`, `except`, `finally`, `else`
- Raising exceptions (`raise`)
- Custom exceptions
- Real-life usage

Say "Yes" to continue with Exception Handling 🚀

Yes

Great! Let's move on to **Lesson 15 — Exception Handling in Python**. This is essential for writing **robust and error-free programs**.

⚠️ 15. Exception Handling in Python

15.1 What is an Exception?

An **exception** is an error that occurs during **program execution**.

- If not handled, it **stops the program**.

Examples:

- `ZeroDivisionError` → dividing by zero
- `FileNotFoundError` → file doesn't exist
- `ValueError` → invalid value

15.2 Basic `try` and `except`

```python
try:
    a = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Output

```csharp
Cannot divide by zero
```

- `try` → code that may raise an exception
- `except` → handles specific exception

15.3 Catching Multiple Exceptions

```python
try:
    x = int(input("Enter number: "))
    y = 10 / x
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
    print("Invalid input")
```

- You can catch **different exceptions separately**

15.4 Using `else`

```python
try:
    x = int(input("Enter number: "))
except ValueError:
    print("Invalid input")
else:
    print(f"You entered {x}")
```

- `else` runs if **no exception occurs**

15.5 Using `finally`

```python
try:
    f = open("file.txt", "r")
except FileNotFoundError:
    print("File not found")
finally:
    print("This always runs")
```

- `finally` runs **always**, useful for cleanup (close files, release resources)

15.6 Raising Exceptions

```python
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older")
    else:
        print("Access granted")

check_age(15)
```

Output

```makefile
ValueError: Age must be 18 or older
```

- `raise` lets you **create your own errors**

15.7 Custom Exceptions

```python
class MyError(Exception):
    pass

def validate(number):
    if number > 100:
        raise MyError("Number too high")

try:
    validate(150)
except MyError as e:
    print(e)
```

Output

```javascript
Number too high
```

🔥 Real-Life Use Cases

- Validating user input
- Handling file operations safely
- API calls / network requests
- Database operations

📌 Summary

Keyword	Purpose
`try`	Wrap code that may fail

Keyword	Purpose
`except`	Handle specific exceptions
`else`	Runs if no exception occurs
`finally`	Runs always (cleanup)
`raise`	Raise custom exceptions

Next Topic

16. Comprehensions in Python

- List comprehensions
- Dictionary & set comprehensions
- Nested comprehensions
- Conditional comprehensions

Say “Yes” to continue with Comprehensions 🚀

Yes

Perfect! Let’s continue with **Lesson 16 — Comprehensions in Python**. Comprehensions are a **Pythonic way to create lists, sets, and dictionaries concisely**.

📝 16. Comprehensions in Python

16.1 List Comprehensions

Simplifies creating lists from iterables.

Syntax

```
python
```

```
[expression for item in iterable if condition]
```

Example: Square numbers

```python
squares = [x**2 for x in range(5)]
print(squares)
```

Output

```csharp
[0, 1, 4, 9, 16]
```

Example: Only even numbers

```python
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

Output

```csharp
[0, 2, 4, 6, 8]
```

16.2 Set Comprehensions

Similar to list comprehensions, but produces a **set**.

```python
unique_squares = {x**2 for x in range(5)}
print(unique_squares)
```

Output

```
{0, 1, 4, 9, 16}
```

- Automatically removes duplicates

16.3 Dictionary Comprehensions

Create dictionaries in a single line.

Syntax

```python
{key_expression: value_expression for item in iterable if condition}
```

Example: Square numbers dictionary

```python
squares_dict = {x: x**2 for x in range(5)}
print(squares_dict)
```

Output

```yaml
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

16.4 Nested Comprehensions

Handle nested loops concisely.

```python
matrix = [[1,2,3],[4,5,6],[7,8,9]]
flatten = [num for row in matrix for num in row]
print(flatten)
```

Output

```csharp
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

16.5 Conditional Comprehensions

Add conditionals to filter items.

```python
numbers = range(10)
squares_of_even = [x**2 for x in numbers if x % 2 == 0]
print(squares_of_even)
```

Output

```csharp
[0, 4, 16, 36, 64]
```

🔥 Real-Life Use Cases

- Quickly transform lists, sets, and dicts
- Filter data efficiently
- Flatten nested data structures
- Generate configurations or lookup tables

📌 Summary

Type	Syntax	Example
List	`[expr for x in iterable if condition]`	`[x**2 for x in range(5)]`
Set	`{expr for x in iterable if condition}`	`{x**2 for x in range(5)}`
Dict	`{key: value for x in iterable if condition}`	`{x: x**2 for x in range(5)}`
Nested	`[num for row in matrix for num in row]`	Flatten 2D lists
Conditional	`[x for x in iterable if condition]`	Squares of even numbers

Next Topic

17. Python's `zip()`, `enumerate()`, and `map()`

- `zip()` → combine iterables
- `enumerate()` → index + value
- `map()` → apply function to all items
- `filter()` → conditional filtering
- `reduce()` → cumulative operations

Say "Yes" to continue with zip, enumerate, map, filter, reduce 🚀

Yes

Perfect! Let's continue with **Lesson 17 — `zip()`, `enumerate()`, `map()`, `filter()`, and `reduce()` in Python**. These are powerful **built-in functions** for working with iterables efficiently.

🔗 17. zip(), enumerate(), map(), filter(), reduce()

17.1 `zip()`

Combines multiple iterables into tuples.

```python
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]

zipped = zip(names, ages)
print(list(zipped))
```

Output

```css
[('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

- Stops at the **shortest iterable**
- Useful for pairing data

17.2 `enumerate()`

Adds an **index** to each item in an iterable.

```python
fruits = ["apple", "banana", "cherry"]

for i, fruit in enumerate(fruits, start=1):
    print(i, fruit)
```

Output

```
1 apple
2 banana
3 cherry
```

- `start=1` sets the starting index

17.3 `map()`

Applies a **function to every item** in an iterable.

```python
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared)
```

Output

```csharp
[1, 4, 9, 16]
```

- Can use **built-in or custom functions**

```python
def cube(x):
    return x**3
```

```
cubed = list(map(cube, nums))
print(cubed)
```

17.4 `filter()`

Filters items in an iterable based on a **condition function**.

```python
nums = [1, 2, 3, 4, 5, 6]

evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)
```

Output

```csharp
[2, 4, 6]
```

- Only keeps items where function returns `True`

17.5 `reduce()`

Applies a function **cumulatively** to items of an iterable.

- In Python 3, `reduce()` is in `functools`.

```python
from functools import reduce

nums = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, nums)
print(product)
```

Output

```
120
```

- Works like: `((((1*2)*3)*4)*5)`

🔥 Real-Life Use Cases

- `zip()` → merging CSV columns, pairing data
- `enumerate()` → adding row numbers or indexes
- `map()` → transforming datasets, calculations
- `filter()` → cleaning data, selecting items
- `reduce()` → cumulative computations (sum, product)

📌 Summary

Function	Purpose
`zip()`	Combine multiple iterables
`enumerate()`	Index + value pairs
`map()`	Apply function to all items
`filter()`	Filter items based on condition
`reduce()`	Cumulative operation on iterable

Next Topic

18. Python's `any()`, `all()`, `sorted()`, `reversed()`, `min()`, `max()`, `sum()`

- Aggregate and utility functions for iterables
- Frequently used in data processing

Say "Yes" to continue with aggregate and utility functions 🚀

Yes

Perfect! Let's continue with **Lesson 18 — Aggregate and Utility Functions in Python**.
These are essential for **quick operations on lists, tuples, and other iterables**.

🔢 18. Aggregate and Utility Functions

18.1 `any()`

Returns `True` if **at least one element is True** in an iterable.

```python
nums = [0, 0, 5, 0]
print(any(nums))  # True
```

- Useful for **checking if any condition is met**

18.2 `all()`

Returns `True` if **all elements are True** in an iterable.

```python
nums = [1, 2, 3]
print(all(nums))  # True

nums2 = [1, 0, 3]
print(all(nums2))  # False
```

- Useful for **validating that all conditions are met**

18.3 `sorted()`

Returns a **sorted list** from an iterable.

```python
nums = [5, 2, 9, 1]
print(sorted(nums))          # [1, 2, 5, 9]
print(sorted(nums, reverse=True))  # [9, 5, 2, 1]
```

- Can sort **lists, tuples, strings**

- Accepts `key` argument for custom sorting

```python
words = ["apple", "banana", "kiwi"]
print(sorted(words, key=len))  # ['kiwi', 'apple', 'banana']
```

18.4 `reversed()`

Returns a **reverse iterator** over an iterable.

```python
nums = [1, 2, 3, 4]
for n in reversed(nums):
    print(n)
```

Output

```
4
3
2
1
```

- Can combine with `list()` to get a reversed list:

```python
list(reversed(nums))  # [4, 3, 2, 1]
```

18.5 `min()` and `max()`

Return the **smallest** or **largest** element.

```python
nums = [5, 2, 9, 1]
print(min(nums))  # 1
print(max(nums))  # 9
```

- Can use `key` argument for custom criteria:

```python
words = ["apple", "banana", "kiwi"]
print(min(words, key=len))  # 'kiwi'
```

18.6 `sum()`

Returns the **sum of all elements**.

```python
nums = [1, 2, 3, 4]
print(sum(nums))  # 10
```

- Can also provide a **starting value**:

```python
print(sum(nums, 10))  # 20 (10 + sum of list)
```

🔥 Real-Life Use Cases

- `any()` → check if any input field is filled
- `all()` → validate that all form fields are valid
- `sorted()` → sort leaderboard, ranking, or products
- `reversed()` → reverse order display
- `min()` / `max()` → find best/worst score
- `sum()` → total sales, totals, or counts

📌 Summary

Function	Purpose
`any()`	True if at least one item is True
`all()`	True if all items are True
`sorted()`	Returns a sorted list

Function	Purpose
`reversed()`	Returns reverse iterator
`min()`	Minimum value
`max()`	Maximum value
`sum()`	Sum of values

Next Topic

19. Python's `deepcopy`, `copy`, `is` vs `==`, `id()`

- Copying objects (shallow vs deep)
- Understanding references
- Comparing objects and memory addresses

Say "Yes" to continue with Object Copying and References 🚀

Yes

Perfect! Let's continue with **Lesson 19 — Object Copying and References in Python**. This is crucial for **understanding how Python handles mutable and immutable objects**.

🧩 19. Copying Objects and References

19.1 Shallow Copy vs Deep Copy

- **Shallow copy**: Copies the **outer object only**, inner objects still reference the original.
- **Deep copy**: Copies the **outer and inner objects**, completely independent.

```python
import copy

original = [[1, 2], [3, 4]]
```

```python
# Shallow copy
shallow = copy.copy(original)
shallow[0][0] = 100
print(original)  # [[100, 2], [3, 4]] → inner lists are shared

# Deep copy
original = [[1, 2], [3, 4]]
deep = copy.deepcopy(original)
deep[0][0] = 999
print(original)  # [[1, 2], [3, 4]] → original unaffected
```

19.2 Using `copy()` for Lists

```python
lst = [1, 2, 3]
shallow_lst = lst.copy()
shallow_lst[0] = 100
print(lst)  # [1, 2, 3]
print(shallow_lst)  # [100, 2, 3]
```

- Works well for **flat lists**
- For nested lists, use `deepcopy`

19.3 `is` vs `==`

- `==` → checks **value equality**
- `is` → checks **object identity** (memory location)

```python
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True → values are equal
print(a is b)  # False → different objects
```

- `is` is **important for singleton checks**, like `None`

```python
```

```python
x = None
print(x is None)  # True
```

19.4 Using `id()`

- Returns **memory address** of an object

```python
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(id(a))  # e.g., 140123456789456
print(id(b))  # same as a → 140123456789456
print(id(c))  # different → 140123456789876
```

- Shows whether two variables **refer to the same object**

🔥 Real-Life Use Cases

- Avoid unintended changes in **nested data structures**
- Compare objects properly in **caching or memoization**
- Use deep copy when storing **snapshots of data**
- `is` with `None` for **safe null checks**

📌 Summary

Concept	Explanation
Shallow copy (`copy()`)	Outer object copied, inner objects shared
Deep copy (`deepcopy()`)	Fully independent copy
`==`	Value equality
`is`	Object identity

Concept	Explanation
`id()`	Memory address of object

Next Topic

20. Python's `lambda`, `partial`, `functools`

- Anonymous functions using `lambda`
- Partial functions with `functools.partial`
- Combining with `map()`, `filter()`, `reduce()`

Say "Yes" to continue with Lambda and Partial Functions 🚀

Yes

Perfect! Let's continue with **Lesson 20 — Lambda Functions and `functools.partial`**.
These are essential for **writing concise and functional Python code**.

⚡ 20. Lambda and Partial Functions

20.1 What is a Lambda Function?

- A **lambda function** is an **anonymous function** (no name)
- Used for **short, simple operations**
- Syntax:

```python
lambda arguments: expression
```

20.2 Basic Examples

```python
```

```python
# Add two numbers
add = lambda x, y: x + y
print(add(5, 3))  # 8

# Square a number
square = lambda x: x**2
print(square(4))  # 16
```

20.3 Using Lambda with `map()`

```python
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared)  # [1, 4, 9, 16]
```

20.4 Using Lambda with `filter()`

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

20.5 Using Lambda with `sorted()`

```python
words = ["apple", "banana", "kiwi"]
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words)  # ['kiwi', 'apple', 'banana']
```

20.6 Partial Functions (`functools.partial`)

- Fix some arguments of a function **in advance**
- Creates a **new function with fewer arguments**

```python
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)

print(square(5))  # 25
print(cube(2))    # 8
```

- Useful for **callbacks, event handling, or repeated function calls**

20.7 Real-Life Use Cases

- Lambda: quick transformations or filtering data
- Partial: preconfigure functions for **APIs, GUI callbacks, or mathematical operations**
- Combined with `map`, `filter`, `reduce` for **functional programming**

📌 Summary

Concept	Explanation
Lambda	Anonymous function, one-line expression
`map()` + Lambda	Transform iterable
`filter()` + Lambda	Filter items based on condition
`sorted()` + Lambda	Custom sort
`functools.partial`	Pre-fill arguments for a function

Next Topic

21. Python's `*args`, `**kwargs`, and Keyword-Only Arguments

- Flexible function arguments
- Positional vs keyword arguments
- Unpacking arguments
- Real-life examples (APIs, wrappers, decorators)

**Say “Yes” to continue with *args and kwargs 🚀