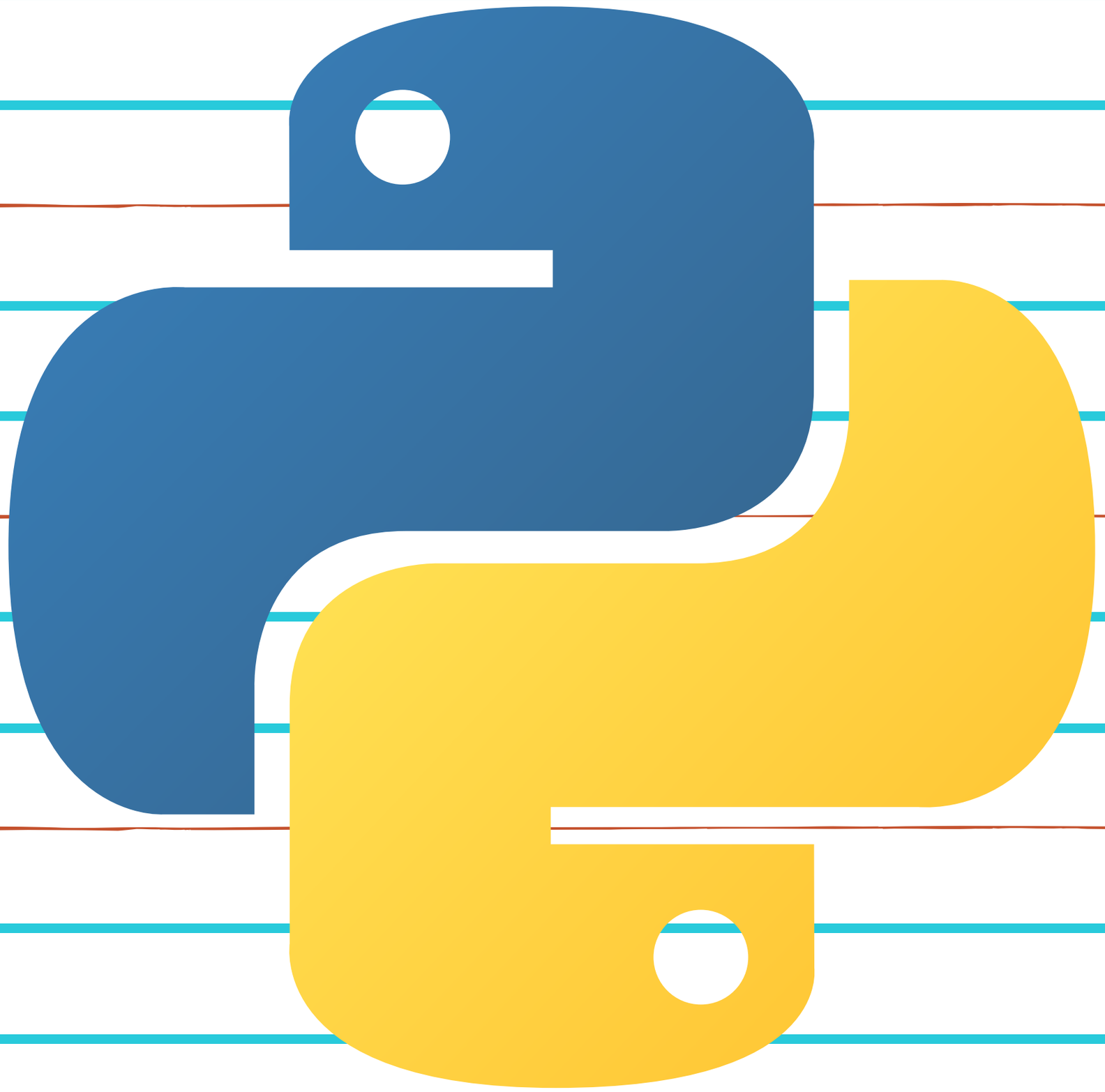
Python

Notes



Deep.T

Why this notes?

1. Python Foundation
   - Data types and variables (int, float, string, bool)
   - Lists, tuples, sets, dictionaries
   - Conditional statements (if-else)
   - Loops (for, while)
   - Functions (def, lambda)
   - Exception handling
   - File I/O

2. Intermediate Python Skills
   - List comprehensions and generator expressions
   - Modules and packages (import, pip)
   - Object-Oriented Programming (classes, inheritance)
   - Decorators and context managers
   - Virtual environments

3. Top 30 Interview Questions

Python Foundations :-

★ **Python :-** Python is a high-level, general-purpose programming language designed by Guido van Rossum and first released in 1991.

Benefits	Limitations
• Easy to learn and read	• Slower execution speed
• Huge community support and rich libraries	• Higher memory usage, not for memory-constrained systems
• Open-source and free	• Sometimes slower for enterprise-scale, high-performance

★ Python Syntax & Indentation

Python syntax is the set of rules that dictate how python code should be structured so the interpreter can run it correctly. The most notable feature is indentation : instead of using braces {} or keywords to define blocks, python uses whitespace at the start of a line.

Example :-

```
if 10 > 5 :
    print ("True")
    print ("Inside if block")
print ("outside the block")
```

★ Data Types :- Python provides several data types

1) Numeric
2) Text
3) Boolean
4) Collections
5) Binary
6) None

7) Numeric Type :-

i) int :- Integer numbers (positive, negative, zero)

Ex :- x = 910

ii) Float :- Decimal (real) numbers

Ex :- x = 3.14

iii) complex :- Numbers with real + imaginary part

Ex :- x = 2 + 6b

2) Text Type :-

Str :- Sequence of characters (text)

Ex :- name = "Python"

3) Boolean Type :-

bool :- Boolean values (True or False)

Ex :- b = True

4) Collections Type :-

i) list :- ordered, mutable collection (allows duplicates)

Ex :- car = ["BMW", "Porsche", "Rolls-Royce"]

ii) tuple :- ordered, immutable collection (allows duplicates)

Ex :- nums = (1, 2, 3, 4)

iii) set :- Unordered, mutable collection (no duplicates)

Ex :- s = {1, 2, 3, 3}

iv) Frozenset :- Immutable version of a set

Ex :- fs = Frozenset ({1, 2, 3})

v) dict :- key-value pairs (unordered, mutable)

Ex:- d = { "name": "JD", "age": 25 }

5) Binary Type :-

i) bytes :- Immutable sequence of bytes

Ex:- b = b"Hello"

ii) bytearray :- Mutable sequence of bytes

Ex:- ba = bytearray([65, 66, 67])

iii) memoryview :- View of byte data without copying

Ex:- mv = memoryview(b"Python")

6) None Type :- Represents "no value" or "null"

Ex:- n = None

* **Data Structure :-** In Python, a data structure is a specialized format for organizing, processing, and storing data in memory to facilitate efficient access and manipulation.

1) List
2) Tuples
3) Sets
4) Dictionaries

1) **List :-** A list is an ordered, mutable, collection of items. Lists can store elements of different data types like integers, strings, floats, etc.

Syntax :-

```
my_list = [1,2,3,4,5]
mixed_list = [1, "apple", 3.14, True]

# Operations
my_list = [1,2,3,4]
my_list.append(5) # [1,2,3,4,5]
my_list.insert(2,99) # [1,2,99,3,4,5]
my_list.remove(3) # [1,2,99,4,5]
my_list.pop() # 5 , [1,2,99,4]
my_list.clear() # []
my_list.sort() # my_list = [1,5,4,3,2] out = [1,2,3,4,5]
my_list.reverse() # [5,4,3,2,1]
my_list.index(3) # 2
my_list.count(2) # [1,2,2,3] , 2
my_list.extend([6,7]) # [5,4,3,2,1,6,7]
```

```
print (3 in my_list) # True
len(my_list) # 7
```

2) Tuple : A tuple is an ordered, immutable collection of items.

Syntax :

```
my_tuple = (1,2,3,4)
single_element = (9,)

# Operations
my_tuple.index(2) # 1
my_tuple.count(2) # (1,2,2,3), output = 2
new_tuple = my_tuple + (4,5) # Concatenation (1,2,2,3,4,5)
my_tuple * 2 # (1,2,2,3,1,2,2,3)
2 in my_tuple # True
```

3) Set : A set is an unordered, mutable, and unique collection of items.

Syntax :

```
my_set = {1, 2, 3, 4}
empty_set = set()

# Operations
my_set.add(5) # {1, 2, 3, 4, 5}
my_set.remove(2) # {1, 3, 4, 5}
my_set.discard(4) # {1, 3, 5}
```

```
my_set.pop()  # {3,5} , 1 removed.
my_set.clear()  # set()
my_set.union(new_set)  # my_set={1,2,3} new_set={2,3}
# {1,2,3}
my_set.intersection(new_set)  # {2}
my_set.difference(new_set)  # {1}
my_set.symmetric_difference(new_set)  # {1,3}
3 in new_set # True
```

4) Dictionaries: A dictionary is an unordered, mutable, key-value pair collection.

Syntax:

```
my_dict = {
    "name" : "PythonBoy",
    "age" : 34,
    "city" : "New York"
}

#Operations
my_dict["name"]  # PythonBoy
my_dict["age"] = 20  # 'age' = 20
my_dict["gender"] = "Male"  # {'name'... 'gender'='Male'}
del my_dict["age"]  # {'name'="PythonBoy", 'city'="New York",
'gender'="Male"}
my_dict.get("city")  # New York
my_dict.keys()  # dict_keys(['name', 'city', 'gender'])
my_dict.values()  # dict_values(["PythonBoy", "New York", "Male"])
```

```
my_dict.items()
# dict_items([('name', 'PythonBoy'), ('city', 'New York'),
('gender', 'Male')])
my_dict.pop("city") # {'name': 'PythonBoy', 'gender': 'Male'}
my_dict.popitem() # {'name': 'PythonBoy'}
my_dict.update({"country": "USA"})
# {'name': 'PythonBoy', 'country': 'USA'}
my_dict.setdefault("city", "Los Angeles")
# {'name': 'PythonBoy', 'country': 'USA', 'city': 'Los Angeles'}
"name" in my_dict # True
```

★ Conditional Statements & Conditional statements are used to perform different actions depending on whether a certain condition is True or False.

1) if
2) if - else
3) if - elif - else

1) if Statement :- This is the simplest form. The code inside the if block runs only if the condition is True.

Syntax :-

```
if condition :
    # code to execute
```

Example :-

```
age = 18
if age >= 18 :
    print (" You are eligible to vote. ")
```

2) if-else Statement :- This is used when you want to provide an alternative block of code if the condition is False.

Syntax :-

```
if condition:
    # code to execute if condition is True
else condition:
    # code to execute if condition is False
```

Example :

```
age = 16
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

3) if-elif-else Statement : This is useful when you have multiple conditions to check.

Syntax :

```
if condition1:
    # code for condition 1
elif condition2:
    # code for condition 2
else:
    # code if none of the conditions are True
```

Example :

```
marks = 85
```

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

* Comparison Operators Used in Conditions:- This are commonly used in if statement:

Operator	Meaning
==	Equal to : if both same
!=	Not equal to : if both are not same
>	Greater than : left > Right
<	Less than : left < Right
>=	Greater than or equal to
>=	Less than or equal to

* Logical Operators:-

Operator	Meaning
and	both conditions must be true
or	at least one conditions true
not	Reverses the result

* Nested if Statements :- You can also place if statements inside another if.

Example :-

```
num = 10
if num > 0:
    if num % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
```

* Ternary (One-line) Conditional Expression :- Python also supports writing conditions in a single line using this Format.

Syntax :-

```
value_if_true if condition else value_if_False
```

Example :-

```
age = 17
message = "Eligible" if age >= 18 else "Not eligible"
print(message)
```

★ **Loops** :- Loops in Python are used to execute a block of code repeatedly until a condition is met or for each item in a sequence

1) For loop 2) while loop

1) **For loop** :- The for loop is used to iterate over a sequence or range, executing a block of code for each item.

Syntax :-

```
For variable in sequence :
    # code block
```

Example :-

```
Fruits = ["apple", "banana", "cherry"]
For fruit in Fruits :
    print(fruit)
```

~~# range~~

```
For i in range(3):
    print (i) # 0 1 2
```

2) **While Loop** :- The while loop runs as long as the specified condition is True.

Syntax:

```
while condition :
    # code block
```

Example :-

```
i = 1
while i <= 3:
    print("Iteration", i)
    i += 1
```

* Nested loop : A loop inside another loop is called a nested loop. Inner loops are executed completely for each iteration of the outer loop.

Example :

```
for i in range(2):
    for j in range(3):
        print(f"i = {i}, j = {j}")
```

output :-

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

* Continue Statement :- The continue statement skips the current iteration and moves to the next iteration of the loop.

Example :-

```
for i in range(5):
    if i==2:
        continue
    print(i)
```

Output :-

```
0
1
3
4
```

* Break Statement :- The break statement stops the loop entirely and exits from it when a condition is met.

Example :-

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output :

```
0
1
2
```

* Else in For Loop :- An else block after a loop is executed when the loop completes normally, meaning it wasn't terminated by a break statement.

Example :-

```
for i in range(3):
```

```
        print(i)
else:
    print("Loop completed without break")
```

Output :-

0
1
2
Loop completed without break

Note :- Same as for, the else block after a while loop runs if the loop wasn't interrupted by a break.

* Pass Statement :- The pass statement is a placeholder that does nothing. It's used when a statement is syntactically required but you don't want any action.

Syntax :-

```
for i in range(3):
    pass
```

* **Functions :** A function is a reusable block of code that performs a specific task. Function help to organize code, avoid repetition and improve readability.

Syntax :

```
def function_name(parameters):
    # code block
    return value # optional
```

* **Parameters :** Parameters are variables listed inside the parenthese of a function definition. They allow you to pass information into functions so that the same function can perform its task with different inputs.

* **Arguments :** Arguments are the actual values you pass to the function when you call it. These values are assigned to the parameters

* **Types of Parameters :**

1) Positional Parameters
2) Default Parameters
3) Keyword Arguments
4) * Arbitrary Arguments (*args)
5) ** Arbitrary Keyword Arguments (**kwargs)

1) Positional Parameters :- The simplest type. Parameters are assigned based on the position in which arguments are passed.

Syntax :-

```
def greet(name, age)
    print(f"Hello {name}, you are {age} y old.")
greet("Python", 25)
```

2) Default Parameters :- A default value is provided in case the argument is not passed.

Syntax :-

```
def greet(name, age=30)
    print(f"Hello {name}, you are {age} years old")
greet("Bob")
greet("Carol", 28)
```

3) Keyword Arguments :- Arguments are passed by name rather than position

Syntax :-

```
def greet(name, age):
    print(f"Hello {name}, you are {age} y old.")
greet(age=22, name="Python")
```

4) * Arbitrary Arguments (args): When you don't know how many arguments will be passed, you use *args to collect them as a tuple.

Syntax:

```
def add(*args):
    return sum(args)
print(add(1,2)) # 3
print(add(1,2,3,4)) # 10
```

5) ** Arbitrary keyword Arguments (kwargs): when you want to accept arbitrary keyword arguments, you use **kwargs to collect them as a dictionary.

Syntax:

```
def details(**kwargs)
    for key, value in kwargs.items():
        print(f"{key}: {value}")
details(name="Eve", age=27, city="New York")
```

* Lambda Function: A lambda function is an anonymous (nameless) function expressed in a single line. It's useful for small tasks where writing a full function would be overkill.

Syntax:

lambda arguments : expression

Example:

```
add = lambda a, b : a+b

result = add(5,3)
print(result) # 8
```

* Built-in Functions :

a) print() :- Display output on the screen.

```
print("Hello, Python")
```

b) input() :- Takes user input as a string.

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

c) len() :- Returns the length of an object like a list, string, or tuple.

```
print(len("Python")) # 6
print(len([1,2,3])) # 3
```

d) range() :- Generates a sequence of numbers.

commonly used in loops.

```
for i in range(1,5):
    print(i)
```

e) type() : Returns the type of an object

```
print(type(5)) # <class 'int'>
```

f) int(), float(), str() : Converts a value to integer, float, or string.

```
num = int("10")
print(num + 8) # 18
```

g) sorted() : Sorts items in a list, tuple, or other iterable.

```
num = [3,1,4,2]
print(sorted(numbers)) # [1,2,3,4]
```

h) max(), min() : Returns the largest or smallest item.

```
num = [5,2,9]
print(max(num)) # 9
print(min(num)) # 2
```

i) sum() : Returns the sum of items in an iterable.

```
num = [1, 2, 3]
print (sum(num)) #6
```

★ Function Widely Used in Companies :

a) map() : Applies a function to all items in an iterable.

Syntax : map(function, iterable)

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

b) filter() : Filters items in an iterable based on a function.

Syntax : filter(function, iterable)

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

c) zip() : Combines multiple iterables into tuples.

Syntax : zip(iterable 1, iterable 2, ...)

```
names = ["Alice", "Bob"]
scores = [85, 92]
result = zip(names, scores) # [('Alice', 85), ('Bob', 92)]
```

VISION

d) enumerate() : Adds an index to iterable items.

Syntax :- enumerate(iterable, start=0)

```
Fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(Fruits):
    print(index, fruit)
```

#output
0 apple
1 banana
2 cherry

e) any(), all() :- any() returns true if any element is true.
all returns true if all elements are true.

Syntax :- any(iterable) all(iterable)

```
print(any([0, 1, 0])) # True
Print(all([1, 1, 1])) # True
```

f) open() :- Opens a file and returns a file object.

Syntax :- open(filename, mode)

```
with open("example.txt", "w") as file:
    file.write("Hello, File!")
```

g) Functools.reduce() : Applies a rolling computation to items in an iterable.

Syntax: From functools import reduce
reduce (function, iterable)

```
From functools import reduce
nums = [1,2,3,4]
result = reduce (lambda x, y: x+y, nums)
print (result)  # 10
```

★ **Exception Handling**: When a program encounters an error during exception. if the exception is not handled, the program will crash and display an error message. Exception handling allows the program to gracefully handle errors and continues or exit with proper messages.

★ **Types of Exceptions**: Python has built-in exceptions like:

	Exception	Description
1)	ZeroDivisionError	Division by zero error
2)	ValueError	Wrong data type or invalid value
3)	TypeError	Operation applied on wrong type
4)	IndexError	Index out of range error
5)	KeyError	Accessing a dictionary with invalid key
6)	FileNotFoundError	File not found
7)	ImportError	Importing a module failed
8)	Exception	Base class. For all exceptions

Notes: You can also create custom exceptions by inheriting from Exception.

```
try:
    value = int(input("Enter number: "))
except ValueError:
    print("Please enter a valid integer.")
```

* Logging Exceptions Instead of Printing : In production code, exceptions are usually logged rather than printed. Python's logging module is used for this.

```
import logging

logging.basicConfig(level=logging.ERROR)
try:
    1/0
except ZeroDivisionError as e:
    logging.error("Division error occurred",
                  exc_info=True)
```

* Using Finally for Cleanup : When working with files, network connections, or databases, finally ensures resources are closed or released.

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError as e:
    logging.error("File not found")
finally:
    file.close()
```

* Context Managers (with statement) : Instead of using try-finally manually, Companies prefer context managers to handle resource

management automatically.

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError as e:
    logging.error("File not found")
```

* Retry Mechanisms: For transient errors (like network issues or temporary server unavailability), companies implement retry logic using libraries like retrying, tenacity, or custom loops.

```
From tenacity import retry, stop_after_attempt,
wait_fixed

@retry(stop=stop_after_attempt(3), wait=wait_fixed
def risky_function():
    print("Trying risky operation")
    1/0

try:
    risky_function()
except ZeroDivisionError:
    print("All retries failed.")
```

* Common libraries Used for Exception Handling in Industry :-

Library	Purpose
logging	Log errors, warnings, info messages
tenacity	Implement retry logic
sentry-sdk	Capture and report errors in production
pytest	Unit testing for exceptions
contextlib	Advanced resource management

* File I/O :- File Input/Output (I/O) is critical because software system need to interact with external data sources - whether it's reading logs, processing user-uploaded files, storing configurations, handing large datasets, or exchanging information between services.

Python's file handling is widely used in:

- Data pipeline (ETL - extract, transform, load processes)
- Configuration management (reading environment settings)
- Logging systems (recording runtime details for debugging)
- Backup scripts (saving snapshots of data)
- File-based database (lightweight storage solutions)

```
with open('config.json', 'r') as file:
    data = file.read()
```

* File Access Modes in Professional Code:-

Mode	Use Case in Industry
'r'	Reading configuration files or logs
'w'	Generating new reports or data dumps
'a'	Appending logs without overwriting existing data

'x' Creating files only if they don't exist - prevents accidental overwrites

'b' Handling non-text files like images, audio, or encrypted content

'+' Read and write scenarios like editing file contents in place

* Handling large file in Production:

```
with open('large-data.csv', 'r') as file:
    for line in file:
        process(line)
```

* Writing Files safely:

```
import tempfile

with tempfile.NamedTemporaryFile('w', delete=False)
    as tmp_file:
    tmp_file.write("Data safely written here")
```

* Working with Structured Files

```
import json
with open('settings.json', 'r') as file:
    settings = json.load(file)
# Modify settings
settings['debug'] = False
with open('settings.json', 'w') as file:
    json.dump(settings, file, indent=4)
```

★ CSV Files :

```
import csv
with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Name', 'Age'])
    writer.writerow(['Alice', 30])
```

★ Error Handling and Logging :

```
try:
    with open('data.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    log.error("File not found: data.txt")
except IOError as e:
    log.error(f"I/O error occurred: {e}")
```

Intermediate Python Skills

* **List Comprehensions:** A list comprehension is a concise way to create lists in Python using a single line of code. It replaces loops and append operation with a compact and readable expression

Syntax:

[expression for item in iterable if condition]

Example 1 - squaring numbers

```
squares = [x**2 for x in range(5)]
print(squares) # [0, 1, 4, 9, 16]
```

Example 2 - Filtering even numbers

```
even = [x for x in range(10) if x%2 == 0]
print(evens) # [0, 2, 4, 6, 8]
```

Example 3 - Working with strings

```
words = ["apple", "banana", "cherry"]
uppercase_words = [word.upper() for word in words]
print(uppercase_words)
# ['APPLE', 'BANANA', 'CHERRY']
```

Example 4 - Nested comprehension

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

★ Generator Expressions : A generator expression is similar to a list comprehension but produces values lazily, one at a time, instead of creating the entire list in memory.

Syntax :

```
(expression for item in iterable if condition)
```

Example 1 - Squaring numbers lazily

```
squares = (x**2 for x in range(5))
print (next (squares)) # 0
print (next (squares)) # 1
```

Example 2 - Sum of squares

```
squares = (x**2 for x in range(10))
total = sum (squares)
print (total) # 285
```

Example 3 - Filtering even numbers

```
evens = [x for x in range(10) if x % 2 == 0]
print(list(evens)) # [0, 2, 4, 6, 8]
```

Note: When to use which?

Use list comprehension when:
- The dataset is small
- You need quick access to all items.
- The operation is simple and you want readable code.

Use generator expressions when:
- The dataset is large
- You want to save memory
- You need to process data lazily or in a pipeli

* **Module in Python :** A module is a file containing Python code - functions, classes, variables, or runnable code - which can be imported and used in other Python programs.

Example : Creating and using a module

`math_utils.py`

```
def add(a,b):
    return a+b
def subtract(a,b):
    return a-b
```

`main.py`

```
import math_utils
result = math_utils.add(5,3)
print("Addition result:", result)
```

Output :- Addition result: 8

* **Mod Package in Python :** A package is a way of structuring Python's namespace using "dotted module name"
  - A package is a directory that contains multiple modules or sub-package.
  - It must contain an `__init__.py` file (even if it's empty) to be recognized as a package in Python versions < 3.3 (optional in newer version, but still good practice)

```
my-package/
    __init__.py
    module1.py
    module2.py
```

```
sub_package/
    __init__.py
    submodule.py
```

```
From my_package import module
From my_package.sub_package import submodule

module.some_function()
submodule.another_function()
```

★ Built-in Modules and Packages :

Module	Description
math	Mathematical functions
datetime	Work with dates and times
os	Interact with the operating system
sys	Access system-specific parameters
json	Work with JSON data

★ Pip :- pip stands for python package installer. It allows you to install external libraries and packages from the python package index (PyPI).

Syntax :

```
pip install package_name
```

Example : pip install pandas

Example :- Checking Installed Packages

```
pip list
```

Example :- Upgrading pip

```
pip install --upgrade pip
```

Virtual Environments :- To avoid conflicts between package versions Python developers often use virtual environments

```
python -m venv myenv
```

Activate it :

windows : `myenv\Scripts\activate`

macOS/Linux : `source myenv/bin/activate`

Note :- any pip install commands will install packages only within this environment.

★ **Object - Oriented Programming** :- Object - Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain both data and functions. It helps structure software by grouping related data (attributes) and behaviors (methods) into reusable components.

1) Class
2) Object
3) Attributes
4) Methods
5) Encapsulation
6) Inheritance
7) Polymorphism
8) Abstraction

1) **Class** :- A class is a blueprint or template for creating objects. It defines attributes (data) and methods (functions) that the object created from the class can use.

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

    def start_engine(self):
        print(f"{self.brand} {self.model}'s
              engine is now running.")
```

2) **Object** :- An object is an instance of a class. Each object can have its own attributes and methods, based on the class.

```
my_car = car("Toyota", "Corolla")
my_car.start_engine()
# Toyota Corolla's engine is now running.
```

3) Attributes :- Attributes are variables that belong to a class or an object. They store the state or properties of the object.

- Instance attributes : Unique to each object
- Class attributes : Shared across all instances.

```
class Car :
    wheels = 4  # class attribute

    def __init__(self, brand, model):
        self.band = brand  #instance attribute
        self.model = model
```

4) Methods :- Methods are functions defined inside a class that describe the behavior of the objects.

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

    def display(self):
        print(f"Brand : {self.brand}, Model :
                {self.model}")
```

vision

5) Encapsulation : Encapsulation is the practice of hiding internal data and requiring all interaction to occur through methods. This prevents direct access and protects data.

- Public attributes/methods can be accessed from outside.
- Private attributes/methods are hidden using underscores.

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

    def get_brand(self):
        return self.__brand
```

6) Inheritance : Inheritance allows a class to inherit attributes and methods from another class.

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

    def move(self):
        print(f"{self.name} is moving.")
class Car(Vehicle):
    def __init__(self, name, brand):
        super().__init__(name)
        self.brand = brand
```

7) Polymorphism : Polymorphism means "many forms" and allows methods to be used in different ways depending on the object.

```
class Dog:
    def speak(self):
        print("Woof!")

class Cat:
    def speak(self):
        print("Meow!")

def animal_sound(animal):
    animal.speak()

dog = Dog()
cat = Cat()

animal_sound(dog)  # Woof!
animal_sound(cat)  # Meow!
```

8) Abstraction : Abstraction hides complex details and shows only necessary parts. Python achieves abstraction using abstract classes and interface (via the abc module)

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
```

```
class Dog(Animal):
    def speak(self):
        print("Woof!")

dog = Dog()
dog.speak()
```

★ How Companies Use OOP :-

1) Code Organization and Maintenance
- In large projects with thousands of files and teams, OOP helps structure code logically.
- Classes and objects allow separation of concerns (eg. Handling users, products, payments in different modules).

2) Reusability Across Projects
- Companies build libraries or frameworks with reusable classes that can be used in different products.

3) Team Collaboration
- by defining standard interfaces and class structures, multiple teams can work independently and integrate seamlessly.

4) Design Patterns
- Companies apply advanced OOP concepts like Factory Pattern, Singleton pattern, Observer Pattern, etc. to solve recurring problems.

5) Scalability and Extensibility
- Applications can grow without breaking existing functionality by extending classes or adding new ones.

6) Security and Data Protection
- Encapsulation ensures sensitive information is not exposed or modified accidentally.

7) Frameworks and Tools
- Popular Python frameworks like Django, Flask, and Fast API are built around OOP principles.
- Even libraries like Numpy, Pandas, and scikit-learn use objects, classes, and methods extensively.

* Decorators and context managers :-

- **Decorators in Python :-** A decorator is a function that takes another function (or method) as input and extends or modifies its behavior without changing its actual code. Decorators are used to wrap a function, add functionality and return it.

```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        print("Wrapper executed before",
              original_function.__name__)
        print("Wrapper executed after",
              original_function.__name__)
        return result
    return wrapper_function

@decorator_function
def display():
    print("Display function ran.")

display()
```

* Context Managers in Python :- A context manager is used to manage resources like files, network connections, or lock by wrapping the setup and teardown code. It ensures that resources are properly

cleaned up after use:

```
class CustomContext:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting the context")
        if exc_type:
            print(f"An error occurred:
                {exc_val}")
        return True # suppresses the exception

with CustomContext() as context:
    print("Inside the context")
    x = 10/0 # this would raise zeroDivision
    error but is handle
```

☆ Python Interview Questions :-

Core Python Basics :-

1) What are Python's main features, and why is it widely used?

Ans :- Python is one of the most popular programming language because of its simplicity and power.

Key features :

- Simple and readable syntax → look close to English
- Interpreted → no need to compile
- Dynamically typed → no need to declare data types explicity (x=10 is valid.)
- Object-Oriented → supports classes, inheritance, polymorphism
- Large standard library & Frameworks → covers web dev (Flask), data science (NumPy, Pandas), AI/ML (TensorFlow, PyTorch)
- Cross-platform → works on Window, Linux, Mac
- Community support → huge open-source community

Why widely used : Easier learning curve.
- Huge ecosystem of libraries
- Used : across field

2) Explain Python's memory management mechanism (Gc, reference counting).

Ans :- Python has automatic memory management handled by the Python Memory Manager.

Key points:

- Reference Counting → Each object keeps a count of references pointing to it.

when count = 0 → object is destroyed

```
a = [1, 2, 3]
b = a    # reference count = 2
del a    # reference count = 1
del b    # reference count = 0 → object removed
```

- Garbage Collector (GC) → Handles circular reference (objects referencing each other).

Ex:- two objects referencing each other will not be deleted by reference counting, so GC cleans them.

- Memory Allocation → Python uses private heaps to store all objects
- Modules used → gc module allows control over garbage collection

3) What are Python's data types? Give example.

Ans:- Python has built-in data types:

- Numeric : int, float, complex

```
x = 10        # int
y = 3.14      # float
z = 2+3j      # complex
```

- Sequence : list, tuple, range

```
l = [1, 2, 3]    # list
t = (1, 2, 3)    # tuple
```

vision

x = range(5) # 0 to 4

- Text : str
  s = "Hello"
- Set Types : set, frozenset
  S = {1,2,3}
  FS = Frozenset([1,2,3]) # immutable set
- Mapping : dict
  d = {"name": "John", "age": 25}
- Boolean : True, False
- None Type : None

4) What is the difference between mutable and immutable objects in Python?

Ans: Mutable -> can be changed after creation.
Example : list, dict, set

```
l = [1,2,3]
l[0] = 100
print(l) # [100,2,3]
```

- Immutable -> cannot be changed after creation.
  Example : int, float, str, tuple, Frozenset

```
s = "Hello"
# s[0] = 'H' error
s = "Hello" # new object
```

5) Explain Python's interpreter and GIL (Global Interpreter Lock).

Ans:- Python Interpreter

- Converts Python code into bytecode (.pyc files).
- Bytecode is executed by the Python Virtual Machine (PVM).
- Most popular interpreter = CPython (written in C)

GIL (Global Interpreter Lock)

- A mutex (lock) that allows only one thread to execute Python bytecode at a time
- Reason: CPython uses reference counting (not thread-safe)
- Effect: limits multi-threaded CPU-bound tasks (like heavy computation)
- Workaround:
  - Use multiprocessing instead of multithreading for CPU-heavy tasks.
  - For I/O-bound tasks (networking, file I/O) threads still help.

Data Structures:-

6) What is the difference between a list, tuple, set and dictionary?

Ans:-

Data structure	Ordered	Mutable	Allows Duplicates	Key Features
List	Yes	Yes	Yes	Dynamic array
Tuple	Yes	No	Yes	Immutable
Set	No	Yes	No	unique elements
Dictionary	Yes (Python 3.7+)	Yes	key duplicate	Stores key-value

7) How do list comprehensions work? Give an example.

Ans:- List comprehension = short way to create lists using a single line.

[expression for item in iterable if condition]

Example:-

```
squares = [x**2 for x in range(5)]
print(squares) # [0, 1, 4, 9, 16]
```

8) How do you use slicing in Python lists/strings?

Ans:- Syntax: sequence[start:end:step]:
work on lists, tuples, strings.

```
nums = [0, 1, 2, 3, 4, 5]
print(nums[1:4]) # [1, 2, 3] (end not included)
print(nums[:3]) # [0, 1, 2]
print(nums[::2]) # [0, 2, 4]
print(nums[::-1]) # [5, 4, 3, 2, 1, 0]

s = "Python"
print(s[0:3]) # "Pyt"
```

9) What is the difference between shallow copy and deep copy?

Ans:- Shallow Copy → Copies the outer object, but references the same inner objects.
Made using copy.copy() or slicing ([:]).
Deep Copy → Creates a new copy of object and all nested objects.

Made using copy.deepcopy()

```
import copy

listy = [[1,2],[3,4]]
shallow = copy.copy(listy)
deep = copy.deepcopy(listy)
listy[0][0] = 99
print(shallow) #[[99,2],[3,4]] (changed!)
print(deep) # [[1,2],[3,4]] (unchanged)
```

10) How do you remove duplicates from a list in Python?

Ans:- Several ways:

Using set (fast, but order lost before Python 3.7):

```
nums = [1,2,2,3,4,4]
unique = list(set(nums))
print(unique) # [1,2,3,4]
```

Using dict.fromkeys (preserves order):

```
nums = [1,2,2,3,4,4]
unique = list(dict.fromkeys(nums))
print(unique) # [1,2,3,4]
```

Using list comprehension:

```
nums = [1,2,2,3,4,4]
unique = []
[unique.append(x) for x in nums if x not in unique]
print(unique) # [1,2,3,4]
```

Functions & Modules :-

Q) Explain the difference between *args and **kwargs

Ans :- Both are used in functions definitions to handle variable numbers of arguments.

*args → collects positional arguments into a tuple.

```
def add(*args):
    return sum(args)
print(add(1,2,3,4)). #10
```

**kwargs → collects keyword arguments into a dictionary.

```
def show_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
show_info(name="Alice", age=25)
#name : Alice
#age : 25
```

Q) What are Python decorators? Give an example use case.

Ans :- Decorator = a function that takes another functions as input and extends/modifies its behavior without changing its code

Implemented using @decorator_name.

Example : Logging function calls

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args}
            {kwargs}")
```

```
        return func(*args, **kwargs)
    return wrapper

@logger
def greet(name):
    print(f"Hello, {name}")

greet("Alice")
# Calling greet with ('Alice',) {}
# Hello, Alice
```

33) What are lambda functions in Python?

Ans:- Anonymous functions (no name), defined with lambda.

lambda arguments : expression

Example:

```
add = lambda x, y: x + y
print(add(3,5)) # 8
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, nums))
print(squares) # [1, 4, 9, 16]
```

34) Explain Python's module vs. package.

Ans:- Module → A single Python file (.py) that contains functions, classes, or variables.

Example:- math.py, my_module.py

Package :→ A collection of modules in a directory with an __init__.py file.

Example folder.

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

Analogy:
Module = chapter in a book.
Package = whole book with chapters.

Q5) How do you handle imports and circular imports in Python?

Ans8- Normal import:

```
import math
From math import sqrt
```

Circular Import Problem
Happens when two modules import each other.

```
# file1.py
import file2
def func1(): pass

#file2.py
import file1
def func2(): pass
```

Note: Restructure code -> put common code in a seperate module
Use local imports -> import inside a function instead of top-level.

OOP in Python

16) What is the difference between class and instance variables?

Ans:- Class Variables
- Shared across all instances of a class.
- Defined inside class, outside methods

Instance Variables
- Unique to each instance (object).
- Defined inside __init__ using self.

```
class Car:
    wheels = 4  # class variable (shared)

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

c1 = Car("Red")
c2 = Car("Blue")

print(c1.wheels, c1.color)  # 4 Red
print(c2.wheels, c2.color)  # 4 Blue
```

17) Explain inheritance in Python with an example.

Ans:- Inheritance = one class (child) derives properties & methods from another (parent).

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

```
class Dog(Animal):
    def speak(self):
        print("Bark!")

d = Dog()
d.speak()  # Bark!
```

Types of inheritance in Python

1) Single 2) Multiple 3) Multilevel
4) Hierarchical 5) Hybrid

Q8) What is method overriding and method overloading

Ans:- Method Overriding

- Child class redefines a method from parent class.

```
class Parent:
    def show(self):
        print("Parent method")
class Child(Parent):
    def show(self): # overriding
        print("Child method")
Child().show() # Child method
```

Method Overloading (same method name, different parameters)

- Python does not support traditional overloading
- Instead: use default arguments or *args.

```
class Math:
    def add(self, a, b=0, c=0):
        return a + b + c
m = Math()
print(m.add(2)) # 2
print(m.add(2,3)) # 5
```

39) What are static method, classmethod, and instance method differences ?

Ans :-

Method Type	Defined with	First Argument	Usage
1) Instance Method	def method(self,)	self → object instance	Access/modify object
2) Class Method	@classmethod	cls → class itself	Access/modify class state
3) Static Method	@staticmethod	No self/cls	Utility Functions

```
class Student :
    school = "ABC School"

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

    def show(self) #instance method
        print(f"Name: {self.name}, School: {self.school}")

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

    @staticmethod
    def greet():
        print("Welcome to Python OOPs")

s = Student("Alice")
s.show() # Instance method
Student.change_school("XYZ School") # class
Student.greet() # static
```

vision

20) How does Python implement encapsulation and polymorphism?

Ans: Encapsulation → binding data & methods together, restricting direct access.

- Python uses naming conventions for access:

Public : var
Protected (convention) : _var
Private (name mangling) : __var

```
class Bank:
    def __init__(self):
        self._balance = 1000  # Protected
        self.__pin = 1234     # private
b = Bank()
print(b._balance)  # Accessible (but conventionally avoid)
#print(b.__pin)  # error
print(b._Bank__pin)  # Accessible via name mangling
```

Polymorphism → same method name, different behavior across classes.

```
class Dog:
    def speak(self): print("Bark")
class Cat:
    def speak(self): print("Meow")
for animal in (Dog(), Cat()):
    animal.speak()
# Bark
# Meow
```

VISION

Exception Handling & File Handling

21) How do you handle exceptions in Python? Explain try-except-else-finally.

Ans: Python handles runtime errors using try-except blocks.

```
try:
    num = int(input("Enter number: "))
    result = 10/num
except ValueError:
    print("Invalid input! Enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful:", result)
finally:
    print("Execution finished.")
```

22) What is the difference between Exception and error in Python?

Ans: Error:

- Problems in the program that usually can't be handle (e.g. syntax error, memory error)
- Example: SytaxError, IndentationError

Exception

- Runtime issue that can be handle with try-except
- Example: ZeroDivisionError, FileNotFoundError.

key difference:
- Errors → mostly due to bad code → must be fixed
- Exceptions → runtime issues → can be caught and handled.

23) How to write and read files in Python?
Ans: Opening a file
```
file = open("test.txt", "w") # modes: r, w, a, rb, wb
```
Writing
```
with open("test.txt", "w") as f:
    f.write("Hello, Python!\n")
```
Reading
```
with open("test.txt", "r") as f:
    content = f.read() # whole file
    # f.readline()     # one line
    # f.readlines()    # list of line
print(content)
```

24) How to Handle memory leaks on file closing safely (with statement)?
Ans: Problem: if you forget file.close(), it causes resource leaks.
```
with open("test.txt", "r") as f:
    data = f.read()
# File automatically closed here
```
- with ensures proper closing, even if an error occurs.

25) What is Python's context manager?

Ans: A context manager resources automatically (setup + cleanup).

- Implements:
  - __enter()__ → executed at the start
  - __exit()__ → executed at the end

Example: (custom context managers)

```
class MyContext:
    def __enter__(self):
        print("Entering context")
        return "Resource"
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting context")

with MyContext() as res:
    print("Using:", res)

# Entering context
# Using: Resource
# Exiting context
```

Advanced Python

26) What are Python generators and how they different from iterators?

Ans: Iterator

- Any object with __iter()__ and __next()__ methods.
- Produces elements one by one.

```
nums = [1, 2, 3]
it = iter(nums)
print(next(it))  # 1
print(next(it))  # 2
```

Generator

- A simpler way to create iterators using yield
- Automatically implements iterator protocol

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

Q2) How does Python's iterator protocol work (iter() and next())?

Ans:- Protocol = ~~rules~~ rules an object must follow to be iterable.

iter(obj) → returns iterator object.

next(iterator) → fetches next item, raises 'StopIteration' at end.

```
nums = [10, 20, 30]
it = iter(nums)

print(next(it))  # 10
print(next(it))  # 20
print(next(it))  # 30
# next(it) -> StopIteration
```

28) Explain Python's multithreading vs. multiprocessing.

Ans:- Multithreading

- Multiple threads in one process.
- Lightweight but limited by GIL
- Good for I/O-bound tasks (network requests)

```
import threading
def task():
    print("Thread running")
t = threading.Thread(target=task)
t.start()
t.join()
```

Multiprocessing

- Multiple separate processes (bypasses GIL).
- True parallelism, good for CPU-bound tasks (math, ML)

```
From multiprocessing import Process
def task():
    print("Process running")
p = Process(target=task)
p.start()
p.join()
```

29) What are Python's built-in data structures (namedtuple, deque, Counter)?

Ans:- namedtuple -> tuple with named fields.

```
From collections import namedtuple
Point = namedtuple("Point", "x y")
p = Point(10, 20)
Print(p.x, p.y)  # 10 20
```

deque -> double-ended queue (fast append/pop from both ends.)

```
From collections import deque
dq = deque([1,2,3])
dq.appendleft(0)
dq.append(4)
print(dq) # deque([0,1,2,3,4])
```

Counter -> counts occurrences of elements

```
from collections import Counter
c = Counter("banana")
print(c) # Counter({'a':3,'n':2,'b':1})
```

30) How do you optimize Python code for performance?

Ans:- General Techniques

1) Use Built-in Functions (fast than manual loops)

```
# slow
result=[]
for x in range(1000):
    result.append(x*2)
# Fast
result = [x*2 for x in range(1000)]
```

2) Use Generators for large datasets -> saves memory

```
def read_large_file(filename):
    with open(filename) as f:
        for line in f:
            yield line
```

3) Use Libraries in C (NumPy, Pandas, cython) → much faster than pure Python.
4) Multiprocessing / Multithreading → distribute work
5) Profiling & optimization
- Use timeit for small code snippets.
- Use cProfile for whole programs.
6) Avoid unnecessary data structures
- Use set membership test (O(1)) instead of list (O(n)).