1. what is python?

---

python is a high level, interpreted, dynamically typed and object oriented programming language known for readability and simplicity.

2. what are python's key features?

```
- easy to learn and read
- interpreted and dynamically typed
- object oriented
- cross platform
- supports functional and procedural programming
```

3. what is the difference between a list and a tuple

---

4. what are python data types?

- int -> integers
- float -> decimal numbers
- `str` -> strings
- bool -> boolean
- list, tuple,set,dict -> collections
- `NoneType` -> represents 'no value'

5. what is the difference between is and ==

---

is -> checks identify (same memory object)
`==` -> check equality(same value)

```
a = [1,2]
b = [1,2]
print(a == b)  # True
```

```python
print(a is b)  # False
```

6. what are python functions and how do you define one

A function is a reusable block of code defined using `def.`

```python
def greet(name):
    return f'my name is {name}'
```

7. what is indentation in python

---

indentation defines code blocks instead of curly breaks {}. typically 4 spaces are used

8. what is None in python?

`None` represents the absence of a value or null value

9. what are `*args` and `**kwargs`

- `*args` -> variable length positional arguments
- `**kwargs` -> variable length keyword arguments

10. what is lambda function?

---

a small anonymous function defined with lambda keyword

```python
square = lambda x: x**2
```

11. What is list Comprehension?

A concise way to create list

```python
squares = [ i**2 for i in range(10)]
```

12. what is the difference between shallow and deep copy?

- shallow copy: creates a new objects but references inner objects.
- deep copy: copies all nested objects too.

```python
import copy
a = [[1,2]]
b = copy.copy(a)
c = copy.deepcopy(a)
```

13. what are decorators?

🧠 What is a Decorator in Python?

➡️ A **decorator** is a **function that modifies or extends the behavior of another function (or method) without changing its code**.

It's like a **wrapper** around a function that adds extra functionality.

✅ Definition

> A decorator is a function that **takes another function as input**, adds some functionality, and **returns a new function**.

📘 Basic Example

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

# Original function
```

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

# Apply decorator
greet = decorator(greet)

greet()
```

Output:

```
Before function call
Hello, Atiar!
After function call
```

⚡ Using `@` Syntax (Syntactic Sugar)

Python allows a shorter way using `@` symbol:

```python
@decorator
def greet():
    print("Hello, Atiar!")

greet()
```

✅ Works exactly like the previous example.

🧩 Real-World Example

Logging decorator:

```python
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Function {func.__name__} called with {args} and {kwargs}")
        return func(*args, **kwargs)
    return wrapper

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

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

Output:

```
Function add called with (5, 3) and {}
8
```

⚙️ Key Points

1. Decorators **don't modify the original function's code**.
2. Can decorate **functions, methods, and even classes**.
3. Common built-in decorators:
    - `@staticmethod`
    - `@classmethod`
    - `@property`

💡 Analogy

Think of a decorator like **gift wrapping**:

- You have a **core gift** (original function)
- You **wrap it** with extra functionality (logging, timing, authentication, etc.)

Do you want me to do that?

decorators modify the behavior of a function or class

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

@decorator
def say_hi():
    print("Hi")

say_hi()
```

14. what is the difference between `@staticmethod` and `@classmethod`

🧠 Difference Between `@staticmethod` and `@classmethod` in Python

Both are **special decorators** in Python classes, but they behave differently.

---

1. `@staticmethod`

- A **static method doesn't take `self` or `cls` as the first argument**.
- It **cannot access instance variables (`self`) or class variables (`cls`)**.
- It's basically a **regular function inside a class** for organizational purposes.

Example:

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

print(Math.add(5, 3))  # 8
```

✅ **Key Points:**

- No access to instance (`self`) or class (`cls`)
- Can be called via class or instance
- Used when the method **doesn't depend on object or class state**

---

2. `@classmethod`

- A **class method takes `cls` as the first argument** (represents the class).
- Can **access/modify class variables** but **not instance variables**.
- Often used for **factory methods** or **alternative constructors**.

Example:

```python
class Person:
    species = "Human"
```

```
    @classmethod
    def show_species(cls):
        print(f"Species: {cls.species}")

Person.show_species()  # Species: Human
```

✅ **Key Points:**

- First parameter is `cls` (the class itself)
- Can access/modify **class-level data**
- Cannot access instance variables directly

⚖️ Comparison Table

Feature	`@staticmethod`	`@classmethod`
First Parameter	None	`cls` (class)
Access Instance Data	❌ No	❌ No
Access Class Data	❌ No	✅ Yes
Can be called on	Class / Instance	Class / Instance
Use Case	Utility function, organization	Factory methods, class-level operations

---

💡 Analogy

- **Static method** → A calculator tool you can use anywhere, doesn't care who owns it.
- **Class method** → Factory or blueprint that can create or modify something for the **class itself**.

---

15. how is memory management in python?

---

python uses automatic garbage collection and reference counting to manage memory

🧠 Memory Management in Python

Python automatically handles memory allocation and deallocation for objects. This means **you don't have to manually manage memory**, unlike languages like C/C++.

Python uses **private heap space**, **garbage collection**, and reference counting to manage memory efficiently.

1️⃣ Python Memory Components

1. **Heap Memory**
   - All Python objects and data structures are stored in a **private heap**.
   - The programmer **cannot access it directly**.
   - Managed internally by the **Python memory manager**.
2. **Stack Memory**
   - Stores **function calls, local variables, and execution context**.
   - Managed automatically using **call stack**.
3. **Code and Data Segments**
   - Python's code objects, constants, and static variables are stored here.

2️⃣ Memory Management Techniques

a) Reference Counting

- Each object keeps a **count of references** pointing to it.
- When reference count becomes **zero**, memory is freed automatically.

```
x = [1, 2, 3]  # reference count = 1
y = x          # reference count = 2
del x          # reference count = 1
del y          # reference count = 0 → memory freed
```

b) Garbage Collection (GC)

- Python uses **automatic garbage collection** to clean memory for objects **with circular references** (not handled by simple reference counting).
- Uses the `gc` **module** to detect and delete unreferenced objects.

```
import gc
print(gc.isenabled())  # Check if garbage collector is enabled
gc.collect()           # Manually run garbage collection
```

c) Dynamic Typing & Object Pooling

- Python **allocates memory dynamically** when objects are created.
- Immutable objects like **small integers and strings** are **interned** (stored in a pool) for efficiency.

3️⃣ Python Memory Manager

Python memory manager handles:

- **Allocation** for objects
- **Deallocation** for unreferenced objects
- **Optimization** for small objects (object pools, free lists)

4️⃣ Key Points

- Python handles memory automatically → easier to code
- Reference counting **cannot handle circular references**, so GC is used
- Large objects or huge data structures → can still cause memory issues
- Use `del`, `gc.collect()`, or memory-efficient structures when needed

💡 Example: Circular Reference

```
class Node:
    def __init__(self):
        self.ref = None

a = Node()
b = Node()
a.ref = b
b.ref = a  # Circular reference

del a
del b
# GC automatically detects and frees memory
```

---

15. Explain mutable and immutable objects with examples?

- mutable: can be changed(list,dist,set)
- immutable: can not be changed(tuple, `str` ,int)

🧠 MUTABLE OBJECTS

Mutable = **can be changed after creation** (in place).

1️⃣ List (mutable)

```
fruits = ["apple", "banana", "cherry"]
print("Before:", fruits)

fruits.append("mango")      # Add item
fruits[1] = "orange"        # Modify item

print("After:", fruits)
```

Output:

```
Before: ['apple', 'banana', 'cherry']
After: ['apple', 'orange', 'cherry', 'mango']
```

✅ List changed → **mutable**.

2️⃣ Dictionary (mutable)

```
person = {"name": "Alice", "age": 25}
print("Before:", person)

person["age"] = 26          # Modify value
person["city"] = "Paris"    # Add new key-value pair

print("After:", person)
```

Output:

```
Before: {'name': 'Alice', 'age': 25}
After: {'name': 'Alice', 'age': 26, 'city': 'Paris'}
```

✅ Dictionary changed → **mutable**.

3️⃣ Set (mutable)

```
numbers = {1, 2, 3}
print("Before:", numbers)

numbers.add(4)
numbers.remove(2)

print("After:", numbers)
```

Output:

```
Before: {1, 2, 3}
After: {1, 3, 4}
```

✅ Set changed → **mutable**.

4️⃣ Bytearray (mutable)

```
data = bytearray([10, 20, 30])
print("Before:", data)

data[1] = 100
print("After:", data)
```

Output:

```
Before: bytearray(b'\n\x14\x1e')
After: bytearray(b'\n d')
```

✅ Bytearray changed → **mutable**.

🧱 IMMUTABLE OBJECTS

Immutable = **cannot be changed after creation**.

1️⃣ Tuple (immutable)

```
coords = (10, 20, 30)
print(coords)

# coords[0] = 100  # ❌ Error: tuple does not support item assignment
```

✅ Tuples cannot be modified.

2️⃣ String (immutable)

```
name = "hello"
print("Before:", name)

# name[0] = "H"  # ❌ Error
new_name = name.replace("h", "H")

print("After:", new_name)
print("Original still:", name)
```

Output:

```
Before: hello
After: Hello
Original still: hello
```

✅ String didn’t change → **immutable**.

3️⃣ Integer (immutable)

```
a = 10
print("Before:", a, "→ ID:", id(a))

a = a + 5
print("After:", a, "→ ID:", id(a))
```

✅ The **ID (memory address)** changed → a new object was created → **immutable**.

4️⃣ Float (immutable)

```
x = 3.14
y = x + 1
print(x, y)
```

✅ Floats create a new object when changed → **immutable**.

5️⃣ Boolean (immutable)

```
flag = True
# flag = not flag creates a new object
```

✅ Booleans don't change — **immutable**.

6️⃣ Frozenset (immutable)

```
fs = frozenset([1, 2, 3])
print(fs)

# fs.add(4)  # ❌ Error: 'frozenset' object has no attribute 'add'
```

✅ Frozenset cannot be changed → **immutable**.

✅ SUMMARY TABLE

Type	Mutable?	Example
`list`	✅ Mutable	`[1, 2, 3]`
`dict`	✅ Mutable	`{"a": 1}`
`set`	✅ Mutable	`{1, 2, 3}`
`bytearray`	✅ Mutable	`bytearray([1,2])`
`tuple`	❌ Immutable	`(1, 2, 3)`
`str`	❌ Immutable	`"hello"`
`int`	❌ Immutable	`10`

Type	Mutable?	Example
`float`	❌ Immutable	`3.14`
`bool`	❌ Immutable	`True`
`frozenset`	❌ Immutable	`frozenset({1,2,3})`

---

16. difference between module and package?

✅ What are Python Modules and Packages?

📦 Python Module

A **module** is a single Python file (`.py`) that contains Python code — such as variables, functions, classes, or runnable code — that you can import and use in another Python script.

🔹 Example:

```
# file: mymodule.py
def greet(name):
    return f"Hello, {name}!"
```

You can import and use it like this:

```
# file: main.py
import mymodule

print(mymodule.greet("Alice"))
```

✅ Key Points:

- One file = one module
- Promotes **code reuse** and **organization**

📁 Python Package

A **package** is a **collection of Python modules** organized in directories that include a special `__init__.py` file.

- The `__init__.py` file makes Python treat the directory as a package.
- It can be empty or include package initialization code.

🔹 Example:

```
myapp/              ← This is a package
├── __init__.py
├── module1.py
└── module2.py
```

You can import like this:

```python
from myapp import module1
module1.some_function()
```

✅ Key Points:

- A **package** helps organize related modules in one place.
- Useful for building large applications or reusable libraries.

🔁 Difference Between Module and Package

Feature	Module	Package
Definition	A single `.py` file	A folder with `__init__.py`
Purpose	Organize code logically	Organize related modules
Usage	`import module`	`import package.module`

---

17. What is an iterator and generator?

🌀 1️⃣ Iterator

🔹 Definition

An **iterator** is an object that allows you to **traverse (loop through)** all the elements of a collection (like a list, tuple, or string) **one at a time**, without needing to know how it's stored.

It implements two special methods:

- `__iter__()` → returns the iterator object itself.
- `__next__()` → returns the next element in the sequence.

🔸 Example: Iterator from a list

```
numbers = [1, 2, 3]
it = iter(numbers)   # Create an iterator object

print(next(it))   # Output: 1
print(next(it))   # Output: 2
print(next(it))   # Output: 3
# print(next(it)) # ❌ Raises StopIteration (no more elements)
```

✅ You can manually control iteration using `next()`.

🔸 In a for loop

When you write:

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

Python **automatically** creates an iterator behind the scenes and calls `next()` until `StopIteration` is raised.

⚡ 2️⃣ Generator

🔹 Definition

A **generator** is a **special type of iterator** that is created using:

- A **function** with the `yield` keyword, **or**
- A **generator expression** (like a list comprehension but with parentheses).

Generators **don't store** all values in memory — they generate items **on the fly**, which makes them very memory-efficient.

🔸 Example: Generator function

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

gen = count_up_to(3)

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

✅ Each time `yield` runs, the function **pauses** and saves its state.
When `next()` is called again, it resumes from where it stopped.

🔸 Generator expression

```
gen = (x**2 for x in range(3))
for val in gen:
    print(val)
```

Output:

```
0
1
4
```

✅ This creates a generator that yields squares one by one — **not stored in memory** like a list.

🧠 Iterator vs Generator: Summary

Feature	Iterator	Generator
How created	Using `iter()` or custom class with `__iter__()` & `__next__()`	Using a function with `yield` or a generator expression
Memory usage	Can be large (stores all elements)	Very low (produces items one by one)
Return type	Any iterable (list, tuple, etc.)	Generator object
Ease of creation	More code	Very simple
Example	`it = iter([1,2,3])`	`gen = (x for x in range(3))`

🧩 Quick analogy

- **Iterator** → Like reading a **book** page by page.
- **Generator** → Like a **machine** that prints the next page only when you ask for it — it doesn't store all pages.

18. What is the difference between `deepcopy` and `copy` modules?

🧠 Copy vs Deepcopy

Both come from Python's built-in `copy` module:

```
import copy
```

🔹 1️⃣ copy.copy() → Shallow Copy

A **shallow copy** creates a **new object**, but it **does not copy the inner (nested) objects**. Instead, it **copies references** to them.

So, changes to nested (inner) objects affect the copy too.

✅ Example:

```python
import copy

list1 = [[1, 2], [3, 4]]
list2 = copy.copy(list1)  # Shallow copy

list2[0][0] = 99          # Change inner element

print("list1:", list1)
print("list2:", list2)
```

Output:

```
list1: [[99, 2], [3, 4]]
list2: [[99, 2], [3, 4]]
```

🧩 Explanation:
Both `list1` and `list2` share the same **inner lists**, so a change in one affects the other.

🔹 2️⃣ copy.deepcopy() → Deep Copy

A **deep copy** creates a **new object** and **recursively copies all inner (nested) objects** as well.

So, the original and the copy are completely **independent**.

✅ Example:

```python
import copy

list1 = [[1, 2], [3, 4]]
list2 = copy.deepcopy(list1)  # Deep copy

list2[0][0] = 99              # Change inner element

print("list1:", list1)
print("list2:", list2)
```

Output:

```
list1: [[1, 2], [3, 4]]
list2: [[99, 2], [3, 4]]
```

🧠 Explanation:
`deepcopy()` creates new inner lists too, so modifying one does **not** affect the other.

🧩 Summary Table

Feature	`copy.copy()` (Shallow)	`copy.deepcopy()` (Deep)
Copies top-level object	✅ Yes	✅ Yes
Copies nested objects	❌ No	✅ Yes
Independent inner objects	❌ No	✅ Yes
Faster	✅ Yes	❌ Slower
Use case	When inner objects won't be modified	When you need a full, independent clone

---

⚙️ Example Visual

If you have:

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

- **Shallow copy:**
  → New outer list, but inner lists are *shared*.
- **Deep copy:**
  → New outer list **and** new inner lists.

✅ In short

- `copy.copy()` → **One-level copy** (outer object only).
- `copy.deepcopy()` → **Full copy** (outer + all nested objects).

19. What is GIL (Global Interpreter Lock)?

🧠 What is GIL (Global Interpreter Lock)?

The **Global Interpreter Lock (GIL)** is a **mutex (mutual exclusion lock)** used by the **CPython** interpreter (the standard and most common Python implementation).

It ensures that **only one thread executes Python bytecode at a time**, even on multi-core processors.

⚙️ Why does the GIL exist?

Python’s memory management (especially **reference counting**) is **not thread-safe** by default.
So, to prevent multiple threads from modifying Python objects at the same time and corrupting memory, the GIL was introduced.

👉 **In simple words:**

> The GIL makes sure that only **one thread runs Python code** at any given moment inside a process.

🧩 Example of GIL in action

Even if you create multiple threads, **only one thread runs Python code at a time** (though threads can still switch rapidly, giving the illusion of parallelism).

```
import threading

def count():
    for i in range(1000000):
        pass

# Create two threads
t1 = threading.Thread(target=count)
t2 = threading.Thread(target=count)

t1.start()
t2.start()
t1.join()
t2.join()

print("Done!")
```

🧱 You might think two threads will run **twice as fast** — but because of the **GIL**, the second thread must wait its turn.
So this won't truly run in parallel on multiple CPU cores.

⚡ When does the GIL matter?

🧮 CPU-bound tasks (heavy computation)

- GIL is a **problem** here.
- Only one thread can execute at a time → No true parallelism.
- Example: numerical loops, image processing, encryption.

🧠 Solution: use **multiprocessing** (separate processes, each with its own GIL) instead of multithreading.

🌐 I/O-bound tasks (waiting for input/output)

- GIL is **not a big problem** here.
- When a thread waits for I/O (like network, disk, or sleep), the GIL is released, so another thread can run.
- Example: web scraping, file downloads, database queries.

🧩 How to bypass the GIL

1. **Use `multiprocessing` module**
   → Each process has its own GIL.

```
from multiprocessing import Process

def task():
    for i in range(1000000):
        pass

p1 = Process(target=task)
p2 = Process(target=task)
p1.start()
p2.start()
p1.join()
p2.join()
```

✅ True parallelism across CPU cores.

2. **Use C extensions or NumPy**
   → Many scientific libraries release the GIL during heavy computation in C.
3. **Alternative Python implementations**
   - **Jython** (Java-based Python) → No GIL
   - **IronPython** (.NET-based) → No GIL
   - **PyPy** → Still has GIL, but more optimized

📋 Summary

Feature	Description
Full form	Global Interpreter Lock
Purpose	Prevent multiple native threads from executing Python bytecode at the same time
Affects	Multi-threaded CPU-bound code
Not an issue for	I/O-bound code
Workarounds	Multiprocessing, C extensions, alternative interpreters
Exists in	CPython (default Python) only

🧠 In short:

> The GIL ensures thread safety in CPython but limits true parallel execution of threads.
> For CPU-heavy tasks → use **multiprocessing**;
> For I/O-heavy tasks → threads are fine.

---

20. Difference between threading and multiprocessing

🧵 Threading

- **Definition:** Runs multiple threads (smaller units of a process) within the same process.

- **Shared Memory:** All threads share the same memory space — global variables, data structures, etc.
- **Parallelism Type:** *Concurrency*, not true parallelism (in CPython).
- **Best For:** I/O-bound tasks — e.g. network requests, file I/O, waiting for input/output.
- **Overhead:** Lightweight — faster to start and switch between threads.
- **Limitations:**
  - Affected by the **Global Interpreter Lock (GIL)** in CPython, meaning only one thread runs Python bytecode at a time.
  - Thread safety issues — must use locks to avoid race conditions.

✅ **Example Use Case:**
Downloading multiple web pages at once, where threads spend time waiting for responses.

⚙️ Multiprocessing

- **Definition:** Runs multiple *processes*, each with its own Python interpreter and memory space.
- **Shared Memory:** Processes do **not** share memory by default — data must be passed via inter-process communication (IPC), such as `Queue`, `Pipe`, or `shared_memory`.
- **Parallelism Type:** *True parallelism* — can use multiple CPU cores simultaneously.
- **Best For:** CPU-bound tasks — e.g. heavy computations, data processing, image processing.
- **Overhead:** Heavier — creating processes and transferring data between them is slower and uses more memory.
- **Advantages:**
  - Avoids GIL limitations.
  - Safer isolation between tasks (less chance of corrupting shared data).

✅ **Example Use Case:**
Performing large matrix calculations or image transformations across multiple cores.

⚖️ Quick Comparison Table

Feature	Threading	Multiprocessing
Execution Model	Multiple threads in one process	Multiple independent processes
Memory Space	Shared	Separate
GIL (in CPython)	Affected	Not affected
Overhead	Low	High

Feature	Threading	Multiprocessing
Best For	I/O-bound tasks	CPU-bound tasks
Communication	Shared memory, locks	Queues, pipes, shared memory
True Parallelism	No (in CPython)	Yes

🧠 Example Task

We'll square a list of numbers and measure how long it takes with:

1. **Threading**
2. **Multiprocessing**

🧵 Using Threading

```
import threading
import time

def square_numbers(numbers):
    for n in numbers:
        n * n  # CPU work

numbers = range(10_000_000)  # 10 million numbers
threads = []

start = time.time()

# Split the numbers into 4 parts
chunk_size = len(numbers) // 4
for i in range(4):
    t = threading.Thread(target=square_numbers, args=(numbers[i*chunk_size:(i+1)*chunk_size],))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("Threading time:", time.time() - start)
```

⚙️ Using Multiprocessing

```
import multiprocessing
import time

def square_numbers(numbers):
    for n in numbers:
        n * n  # CPU work

numbers = range(10_000_000)  # 10 million numbers
processes = []

start = time.time()

# Split the numbers into 4 parts
chunk_size = len(numbers) // 4
for i in range(4):
    p = multiprocessing.Process(target=square_numbers, args=
(numbers[i*chunk_size:(i+1)*chunk_size],))
    processes.append(p)
    p.start()

for p in processes:
    p.join()

print("Multiprocessing time:", time.time() - start)
```

⚖️ What Happens When You Run This

Version	Expected Result	Why
🧵 **Threading**	Takes nearly the same time as a single-threaded version	The **GIL** prevents true parallel CPU execution — only one thread runs at a time
⚙️ **Multiprocessing**	Much faster on multi-core CPUs	Each process runs on a separate core — **true parallelism**

💡 Tip

If your task is **I/O-bound** (e.g., waiting for web requests or reading files), **threading** is often better.

If your task is **CPU-bound** (e.g., calculations, image processing), **multiprocessing** is the right choice.

---

21. What is monkey patching?

Changing or extending a class or module at runtime

```
import math

math.pi = 3
```

22. What are Python modules and packages?

📦 Python Module

- A **module** is simply a single `.py` file containing Python code — functions, classes, variables, or runnable code.
- It helps organize code logically and allows you to reuse it by importing.
- You can import your own modules or built-in ones.

Example:

```
# my_module.py
def greet(name):
    return f"Hello, {name}!"
```

Then in another file:

```
import my_module

print(my_module.greet("Alice"))  # Output: Hello, Alice!
```

📂 Python Package

- A **package** is a way to organize **multiple modules** into a directory hierarchy.

- It’s basically a folder containing Python modules **and** a special file named `__init__.py` (can be empty or contain initialization code).
- This `__init__.py` file tells Python that this directory should be treated as a package.
- Packages let you organize related modules together under a namespace.

Example package structure:

```
my_package/
    __init__.py
    module1.py
    module2.py
```

You can import from the package like this:

```
from my_package import module1
from my_package.module2 import some_function
```

Quick Summary

Term	What it is	Example
Module	A single Python `.py` file	`math.py`, `my_module.py`
Package	A folder with modules + `__init__.py`	`numpy/`, `my_package/`

---

23. What is `__init__.py` used for?

What is `__init__.py`?

- It's a special Python file that **makes a directory into a Python package**.
- Without this file, Python **won't recognize the folder as a package**, and you won't be able to import modules from it using the package syntax.
- It can be **empty** or contain initialization code for the package.

Why is it used?

1. **Package Initialization**
   When you import a package, Python executes the code in `__init__.py` first. This allows you to set up package-level variables or import `submodules` automatically.
2. **Control What's Imported**
   You can control what's exposed when you do `from package import *` by defining an `__all__` list inside `__init__.py`.
3. **Namespace Management**
   Helps organize your modules and sub-packages neatly under the package namespace.

Example

Say you have this folder structure:

```
my_package/
    __init__.py
    module1.py
```

- If `__init__.py` contains:

```python
from .module1 import some_function
```

- Then you can import `some_function` directly like this:

```python
from my_package import some_function
```

In summary:

Purpose	Details
Mark a directory as package	Makes Python treat folder as a package
Initialization code	Runs setup code when package is imported
Manage imports	Control what symbols the package exposes

---

24. What are dunder (magic) methods?

What are Dunder (Magic) Methods?

- They let you **customize the behavior of your classes**.
- Python automatically calls them in certain situations—like when you use operators (`+` , `*` , `==`), convert to strings, or create objects.
- They make your objects behave like built-in types.

Common Examples

Dunder Method	What It Does	When It's Called
`__init__(self, ...)`	Object constructor (initializer)	When creating a new instance
`__str__(self)`	String representation (user-friendly)	When you call `str(obj)` or `print(obj)`
`__repr__(self)`	Official string representation	In the interactive interpreter or `repr(obj)`
`__add__(self, other)`	Defines behavior for `+` operator	When doing `obj1 + obj2`
`__len__(self)`	Defines behavior for `len()`	When calling `len(obj)`
`__eq__(self, other)`	Defines behavior for equality `==`	When comparing `obj1 == obj2`
`__getitem__(self, key)`	Access item via indexing (`obj[key]`)	When accessing elements by key/index

---

Why Use Them?

- They let you make your classes **intuitive and `pythonic`**.
- They enable **operator overloading**, so your objects can work naturally with operators.
- They allow your classes to integrate smoothly with Python's syntax and built-in functions.

Example: Custom Class with `Dunder` Methods

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

```python
    def __str__(self):
        return f"Point({self.x}, {self.y})"

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

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

p1 = Point(1, 2)
p2 = Point(3, 4)

print(p1)           # Output: Point(1, 2)
print(p1 + p2)      # Output: Point(4, 6)
print(p1 == p2)     # Output: False
```

Example: A `Book` class with dunder methods

```python
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        # User-friendly string representation
        return f"'{self.title}' by {self.author}"

    def __repr__(self):
        # Official string representation (useful for debugging)
        return f"Book(title={self.title!r}, author={self.author!r}, pages=
{self.pages})"

    def __len__(self):
        # Return number of pages when len() is called
        return self.pages

    def __eq__(self, other):
        # Define equality: same title and author means books are equal
        return self.title == other.title and self.author == other.author

# Create two book instances
book1 = Book("1984", "George Orwell", 328)
book2 = Book("1984", "George Orwell", 328)
```

```python
print(book1)              # Uses __str__: '1984' by George Orwell
print(repr(book1))        # Uses __repr__: Book(title='1984',
author='George Orwell', pages=328)
print(len(book1))         # Uses __len__: 328
print(book1 == book2)     # Uses __eq__: True
```

What happens here?

- `__init__`: Initializes the book's attributes.
- `__str__`: Defines what `print(book1)` shows.
- `__repr__`: Shows detailed info useful for debugging.
- `__len__`: Allows `len(book1)` to return number of pages.
- `__eq__`: Compares two books for equality.

---

25. What is the difference between `@property` and normal methods?

🛠️ Normal Methods

- You **call** them with parentheses: `obj.method()`.
- They can take arguments and perform actions.
- Typically used when you want to perform a task or computation explicitly.

```python
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.1416 * (self.radius ** 2)

c = Circle(5)
print(c.area())  # You have to call the method with ()
```

🌟 `@property` Decorator

- Allows you to define a **method that acts like an attribute**.

- You **access it without parentheses**: `obj.attribute`.
- Useful when you want to **calculate or manage an attribute dynamically**, but keep a simple syntax.
- Makes your code cleaner and helps **encapsulate data**.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return 3.1416 * (self.radius ** 2)

c = Circle(5)
print(c.area)  # Accessed like an attribute, no ()
```

Key Differences

Aspect	Normal Method	`@property` Method
Syntax to access	Needs parentheses `()`	Accessed like an attribute
Use case	Actions or computations that may require parameters	Computed attributes, encapsulated access
Readability	Clear it's a method call	Cleaner syntax, looks like a data attribute

---

Why use `@property`?

- To **hide implementation details**.
- To **compute values on the fly** but allow users to use simple attribute access.
- To make your class **interface cleaner and more intuitive**.

Bonus: You can also have setters and `deleters` with `@property`:

```
class Circle:
    def __init__(self, radius):
```

```
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value
```

---

27. Explain exception handling in Python.

What is Exception Handling?

- Exceptions are **errors that occur during program execution** (like dividing by zero, file not found, or accessing invalid indexes).
- **Exception handling** lets you gracefully manage these errors instead of crashing the program.
- It helps you **detect, catch, and respond** to errors to keep your program running smoothly.

Key Concepts

Term	Meaning
Exception	An error that occurs during execution
Try block	Code you want to "try" that might cause an error
Except block	Code that runs if an exception happens
Else block	Runs if no exception occurs in the try block
Finally block	Code that runs no matter what (cleanup)

Basic Syntax

```python
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code to handle the exception
    print("Oops! You can't divide by zero.")
else:
    # Code that runs if no exception was raised
    print("Division succeeded:", result)
finally:
    # Code that runs no matter what
    print("This runs whether or not an exception occurred.")
```

How It Works

- Python tries to execute the code in the **try block**.
- If an error occurs, Python looks for a matching **except block** to handle that specific exception.
- If no error occurs, the **else block** runs.
- The **finally block** always runs, even if an exception wasn't caught (useful for cleanup like closing files).

Catching Multiple Exceptions

```python
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
else:
    print("Result is", result)
```

Raising Exceptions Manually

You can also raise exceptions intentionally using `raise`:

```python
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
```

```python
    print(f"Age set to {age}")

set_age(-5)  # This raises ValueError
```

Summary

Block	Purpose
`try`	Code that might throw an exception
`except`	Handle specific exceptions
`else`	Run if no exceptions occur
`finally`	Always runs (cleanup)

---

28. What is virtual environment (venv)?

What is a Virtual Environment (`venv`)?

- A **virtual environment** is an isolated Python environment that lets you manage packages **separately** from your system-wide Python installation.
- It creates a **self-contained directory** with its own Python interpreter and package folder.
- This means you can install different versions of packages for different projects **without conflicts**.

Why Use a Virtual Environment?

1. **Dependency Isolation:**
   Different projects might need different versions of the same package. `venv` keeps them separate.
2. **Avoid System Pollution:**
   Installing packages globally can mess with system tools or other projects.
3. **Reproducibility:**
   You can create a `requirements.txt` file listing your project's dependencies, making it easier to share and reproduce environments.

How to Create and Use a Virtual Environment

1. **Create a virtual environment:**

```
python -m venv myenv
```

This creates a folder `myenv/` with the isolated Python environment.

2. **Activate the virtual environment:**

- On **Windows:**

```
myenv\Scripts\activate
```

- On **macOS/Linux:**

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

3. **Install packages inside the venv:**

```
pip install requests
```

4. **Deactivate the environment when done:**

```
deactivate
```

Summary

Feature	Description
Isolation	Keeps project dependencies separate
Own Python interpreter	Independent Python version if needed
Package management	Install packages locally per project
Easy to share	Use `requirements.txt` to recreate

---

29. Explain Python’s memory model and garbage collection

Python’s Memory Model

1. Memory Management Basics

- Python manages memory automatically — you don’t manually allocate or free memory like in languages such as C.
- When you create objects (variables, lists, etc.), Python allocates memory on the **heap**.
- The **Python memory manager** handles the allocation and deallocation of this memory.

2. Reference Counting

- Python primarily uses **reference counting** to keep track of how many references point to an object.
- Every object has a **reference count** — the number of places in your program that refer to it.
- When an object’s reference count drops to zero (no references), Python immediately frees the memory.

Example:

```
a = [1, 2, 3]  # reference count for this list increases to 1
b = a          # reference count increases to 2
del a          # reference count decreases to 1
del b          # reference count drops to 0, list is deallocated
```

3. Memory Pools and Arenas

- To optimize memory usage, Python uses a **private heap** with **memory pools** managed by the **Python allocator**.
- Small objects are allocated in pools to reduce fragmentation and overhead.

Garbage Collection (GC)

Why is GC needed?

- Reference counting **alone can’t handle cyclic references** (objects referencing each other).
- Cycles cause memory leaks because reference counts never drop to zero.

How Python’s GC works

- Python’s **garbage collector** is a cyclic GC that complements reference counting.
- It detects groups of objects that reference each other but are no longer reachable from the program.
- Uses a **generational approach** with three generations (0, 1, 2) to optimize collection frequency.

Key points about Python’s GC:

Feature	Description
Primary method	Reference counting
Handles cycles	Generational cyclic garbage collector
Generations	Objects promoted from gen0 → gen1 → gen2 if they survive collections
Trigger	GC runs automatically based on thresholds or can be manually triggered via `gc` module

---

Using the `gc` Module

You can interact with Python’s garbage collector:

```
import gc

# Manually trigger garbage collection
gc.collect()

# Disable GC (not recommended usually)
gc.disable()
```

Summary

- **Python memory management** is automatic, mostly using **reference counting**.
- **Garbage collector** handles cycles that reference counting misses.
- Python optimizes performance with **memory pools** and a **generational GC**.

---

30. What is the difference between `@staticmethod`, `@classmethod`, and normal method?

---

1. Normal Instance Method

- The most common method type.
- Automatically receives the instance (`self`) as the **first argument**.
- Can access and modify **instance attributes** and **other methods** via `self`.

```python
class MyClass:
    def instance_method(self):
        print(f"Called instance_method of {self}")

obj = MyClass()
obj.instance_method()  # Implicitly passes obj as self
```

---

2. `**@staticmethod**`

- Does **not receive** `self` or `cls` automatically.
- Behaves like a **regular function inside the class namespace**.
- Cannot access instance (`self`) or class (`cls`) data.
- Useful for utility functions related to the class but independent of instance or class data.

```python
class MyClass:
    @staticmethod
    def static_method():
        print("Called static_method")

MyClass.static_method()  # No self or cls required
```

---

3. @classmethod

- Automatically receives the **class** (`cls`) as the first argument instead of `self`.
- Can access or modify class state that applies across all instances.
- Often used as **alternative constructors** or methods that affect the class as a whole.

```python
class MyClass:
    class_var = 0

    @classmethod
    def class_method(cls):
        print(f"Called class_method of {cls}")
        cls.class_var += 1

MyClass.class_method()
print(MyClass.class_var)  # Outputs: 1
```

Summary Table

Method Type	First Argument	Can Access Instance Attributes?	Can Access Class Attributes?	Use Case
Normal Method	`self`	Yes	Yes	Instance-specific behavior
`@staticmethod`	None	No	No	Utility functions related to class
`@classmethod`	`cls`	No	Yes	Factory methods, class-wide behavior

Quick Example Combining All Three

```python
class Person:
    species = "Homo sapiens"

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

    def say_hello(self):            # Normal method
        print(f"Hi, I'm {self.name}")

    @staticmethod
    def is_adult(age):              # Static method
        return age >= 18

    @classmethod
```

```python
    def species_name(cls):          # Class method
        return cls.species

p = Person("Alice")
p.say_hello()                      # Hi, I'm Alice
print(Person.is_adult(20))         # True
print(Person.species_name())       # Homo sapiens
```

---

31. What is the difference between `return` and `print`?

✅ Difference Between `return` and `print` in Python

Both `return` and `print()` are commonly used in Python functions, but they serve **very different purposes**.

🔁 `return`

- **Used inside functions** to **send data back** to the caller.
- It **exits** the function and hands a value back to where the function was called.
- The returned value can be **stored in a variable**, **used in expressions**, or **passed to other functions**.

🔹 Example:

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

result = add(3, 4)
print(result)  # Output: 7
```

✔ `return` gives us the value `7`, which we then print or reuse.

🖨️ `print()`

- Used to **display output to the screen (console)**.
- It is for **humans to read**, not for program logic.

- It **does not affect** the program's logic or return any value.

🔹 Example:

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

result = add(3, 4)  # This prints: 7
print(result)       # This prints: None
```

⚠️ `result` is `None` because the function didn't return anything.

🔍 Summary Table

Feature	`return`	`print()`
Purpose	Sends value from function	Displays info to the console
Returns Value?	✅ Yes	❌ No
Used in Logic?	✅ Yes (affects code behavior)	❌ No (for display only)
Reusable?	✅ Returned value can be reused	❌ Printed value is not reusable

---

💡 Tip:

Use `return` when you want to **use the result** later in your program.
Use `print()` when you want to **show the result** to the user or for debugging.

---

32. How is a list different from a set in Python?

✅ Difference Between List and Set in Python

Both **lists** and **sets** are built-in data types in Python that store collections of items — but they behave differently.

🧾 1. List

- An **ordered**, **indexed**, and **mutable** collection.
- **Allows duplicates**.
- Defined using **square brackets**: `[]`

🔹 Example:

```
my_list = [1, 2, 3, 2, 4]
print(my_list[0])     # Output: 1 (indexing allowed)
```

🔘 2. Set

- An **unordered**, **unindexed**, and **mutable** collection.
- **Does not allow duplicates**.
- Defined using **curly braces**: `{}`

🔹 Example:

```
my_set = {1, 2, 3, 2, 4}
print(my_set)          # Output: {1, 2, 3, 4} (duplicates removed)
```

🔍 Key Differences

Feature	List	Set
Ordered	✅ Yes	❌ No (unordered)
Indexed	✅ Yes (can access by index)	❌ No indexing
Duplicates Allowed	✅ Yes	❌ No
Mutable	✅ Yes	✅ Yes
Syntax	`[1, 2, 3]`	`{1, 2, 3}`
Use Case	When order matters	When uniqueness matters

📌 When to Use Which?

- Use a **list** when:
  - You need to maintain order
  - Duplicates are allowed

    - You need to access items by position
- Use a **set** when:
    - You want only **unique** items
    - You want to do set operations like **union**, **intersection**, etc.

---

33. How do you iterate over a dictionary?

✅ How to Iterate Over a Dictionary in Python

A **dictionary** in Python stores **key-value pairs**, and there are several ways to loop through it depending on what you want to access: keys, values, or both.

🔹 1. Iterate Over Keys (Default Behavior)

```
my_dict = {"a": 1, "b": 2, "c": 3}

for key in my_dict:
    print(key, my_dict[key])
```

✅ Output:

```
a 1
b 2
c 3
```

🔹 2. Iterate Over Keys (Explicitly)

```
for key in my_dict.keys():
    print(key)
```

🔹 3. Iterate Over Values

```
for value in my_dict.values():
```

```python
    print(value)
```

✅ Output:

```
1
2
3
```

🔹 4. Iterate Over Key–Value Pairs

```python
for key, value in my_dict.items():
    print(f"{key} => {value}")
```

✅ Output:

```
a => 1
b => 2
c => 3
```

🔹 5. Using `enumerate()` with Dictionary (rarely needed)

If you need indexes along with keys:

```python
for index, key in enumerate(my_dict):
    print(index, key, my_dict[key])
```

🔍 Summary

Method	What it iterates over
`for key in dict`	Keys
`for key in dict.keys()`	Keys
`for value in dict.values()`	Values
`for key, value in dict.items()`	Key–Value pairs

---

standard question and answer

1. what is pep 8 and what is it important

PEP 8 stands for **Python Enhancement Proposal 8**, and it is the **official style guide for writing Python code**.

It defines **rules and best practices** for how Python code should be formatted — so that the code is **clean, consistent, and easy to read** for everyone.

🧠 What PEP 8 Is

PEP 8 is a **document** that gives guidelines on:

- Code layout
- Naming conventions
- Indentation and spacing
- Imports
- Line length
- Comments and documentation

It's basically a **coding standard** that helps Python developers write code in a uniform style.

📘 Example: PEP 8 Rules

Topic	Bad Code ❌	Good Code ✅	Rule
Indentation	`if x>5:print("Yes")`	`if x > 5:\n print("Yes")`	Use 4 spaces per indentation level
Line Length	(Very long line in one line)	Break long lines (< 79 chars)	Keep lines short for readability
Variable Naming	`VarOne = 10`	`var_one = 10`	Use `snake_case` for variables
Class Naming	`class student:`	`class Student:`	Use `PascalCase` for class names
Imports	`import os, sys`	`import os\nimport sys`	One import per line
Whitespace	`x=5+3`	`x = 5 + 3`	Use spaces around operators
Comments	None	`# Explain what the code does`	Use comments to clarify code

💡 Why PEP 8 Is Important

1. ✅ **Improves Readability**
   Makes code easy for others (and your future self) to understand.
2. 🔄 **Maintains Consistency**
   When everyone follows the same style, all Python code looks familiar.
3. 🧩 **Reduces Errors**
   Clean and structured code helps prevent logical and syntax mistakes.
4. 🤝 **Eases Collaboration**
   Multiple developers can work on the same project smoothly.
5. 🧹 **Professional Standard**
   Following PEP 8 shows you're a disciplined, professional Python developer.

🧰 Tools That Help You Follow PEP 8

- `flake8` → checks for style errors
- `black` → automatically formats your code
- `pylint` → checks code quality + style
- `autopep8` → automatically fixes PEP 8 issues

---

show 8 rule for pep

Here are **8 important PEP 8 rules** every Python programmer should know 👇

🧩 1. Indentation

➡️ Use **4 spaces** per indentation level (no tabs).

```
if x > 5:
    print("Big number")
```

✍️ 2. Line Length

➡️ Keep each line **under 79 characters**.
This helps your code look neat on all screens.

🔤 3. Blank Lines

➡️ Use blank lines to separate **functions, classes**, and **logical sections** of code.

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


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

🧠 4. Imports

➡️ Each import should be on a **separate line** and placed **at the top** of the file.

```
import os
import sys
```

💬 5. Naming Conventions

Type	Style	Example
Variable / Function	`snake_case`	`user_name`, `get_data()`
Class	`PascalCase`	`StudentInfo`
Constant	`UPPER_CASE`	`PI = 3.14`

⚙️ 6. Whitespace Around Operators

➡️ Use **spaces** around operators and after commas.

```
x = a + b
print(x, y)
```

❌ Not good → `x=a+b`

🧾 7. Comments

➡️ Write comments that explain *why* something is done, not *what*.

```
# Calculate average marks
average = total / count
```

🧱 8. Function and Class Definitions

➡️ Leave **two blank lines** before defining a new function or class.

```
class Person:
    pass


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

---

2. what are key word in python

✅ **Python Keywords** are **reserved words** that have **special meaning** in the Python language.
You **cannot use** them as variable names, function names, or identifiers.

They are the **core building blocks** of Python syntax (used for conditions, loops, functions, classes, etc.)

🧠 👉 Definition:

> Keywords are predefined words in Python that define the syntax and structure of the language.

🧾 List of Python Keywords (as of Python 3.12)

Category	Keywords
Logical / Boolean	`True`, `False`, `None`

Category	Keywords
Conditional	`if`, `elif`, `else`
Loops	`for`, `while`, `break`, `continue`
Function Related	`def`, `return`, `lambda`, `yield`
Class & Object Related	`class`, `self`, `del`
Import & Module	`import`, `from`, `as`
Exception Handling	`try`, `except`, `finally`, `raise`, `assert`
Variable Scope / Global	`global`, `nonlocal`
Logical Operators	`and`, `or`, `not`, `in`, `is`
Control Flow	`pass`, `break`, `continue`
Async Programming	`async`, `await`
Miscellaneous	`with`, `yield`, `match`, `case` (Python 3.10+)

💡 Example Usage

```
if True:
    for i in range(3):
        print(i)
else:
    pass
```

⚠️ Note:

- Keywords are **case-sensitive** → `True` ≠ `true`
- You can check all Python keywords using:

```
import keyword
print(keyword.kwlist)
```

---

3. python build in datatypes:

✅ **Built-in Data Types in Python** are the **predefined types** that Python provides to store and manipulate different kinds of data — like numbers, text, lists, etc.

They are the **foundation** of all Python programs.

🧠 👉 Definition:

> Built-in data types are the basic types of values that Python can handle directly without any import or library.

🧾 🔹 Main Categories of Python Built-in Data Types

Category	Data Types	Example	Description
1. Numeric	`int`, `float`, `complex`	`a = 10`, `b = 3.14`, `c = 2 + 3j`	Store numbers — integers, decimals, or complex numbers
2. Sequence	`str`, `list`, `tuple`, `range`	`'Hello'`, `[1,2,3]`, `(4,5,6)`, `range(5)`	Ordered collections of items
3. Mapping	`dict`	`{"name": "Atiar", "age": 21}`	Key–value pairs (like a real-world dictionary)
4. Set	`set`, `frozenset`	`{1, 2, 3}`, `frozenset({1,2})`	Unordered collections of unique items
5. Boolean	`bool`	`True`, `False`	Logical values (used in conditions)
6. Binary	`bytes`, `bytearray`, `memoryview`	`b"ABC"`, `bytearray(5)`	Store binary data (for files, images, etc.)
7. None Type	`NoneType`	`None`	Represents “nothing” or “no value”

📘 Examples

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

# Sequence
name = "Atiar"  # str
nums = [1, 2, 3] # list
coords = (4, 5)  # tuple
r = range(5)     # range
```

```
# Mapping
student = {"name": "Atiar", "age": 21}

# Set
colors = {"red", "green", "blue"}

# Boolean
is_active = True

# None
result = None
```

💡 Check Data Type:

You can check any variable's type using:

```
print(type(x))
```

---

what is the difference between parameters and arguments

🧠 Definition

Term	Meaning	Example
Parameter	A variable **defined in the function header** — acts as a placeholder.	`def add(x, y):` → here `x` and `y` are **parameters**
Argument	The **actual value** you pass to the function when calling it.	`add(5, 3)` → here `5` and `3` are **arguments**

📘 Example

```
def greet(name):    # 'name' is a parameter
    print("Hello,", name)

greet("Atiar")      # "Atiar" is an argument
```

🔹 **Parameter** → variable inside the function definition
🔹 **Argument** → actual value you send during function call

⚖️ In Short:

Comparison	Parameter	Argument
Location	Inside function definition	Inside function call
Purpose	Placeholder for data	Real data passed to function
When used	When defining function	When calling function

💡 Analogy:

Think of a function like a **math formula**:
👉 `f(x) = x + 2`
Here `x` is the **parameter**,
and when you do `f(3)` → `3` is the **argument**.

---

there are 5 main types of arguments that you can pass to a function.

Let's go through them one by one with easy examples 👇

🧾 1. Positional Arguments

➡️ Arguments are passed **in order**, matching the function's parameters **by position**.

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

add(5, 3)      # a=5, b=3
```

✅ **Output:** `8`
⚠️ Order matters! If you change it → result changes.

🧾 2. Keyword Arguments

➡️ You specify **which parameter** each argument belongs to by name.

```
def student(name, age):
    print(name, age)

student(age=21, name="Atiar")
```

✅ **Output:** `Atiar 21`
🔹 Order doesn't matter here since you name the arguments.

🧾 3. Default Arguments

➡️ You can give a **default value** to a parameter.
If the caller doesn't pass that argument, the default is used.

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

greet()           # Uses default
greet("Atiar")    # Uses given value
```

✅ **Output:**

```
Hello, Guest
Hello, Atiar
```

🧾 *_4. Variable-Length Arguments (_args)__

➡️ Use `*args` when you **don't know** how many positional arguments will be passed.

```
def add_all(*numbers):
    print(sum(numbers))

add_all(1, 2, 3, 4, 5)
```

✅ **Output:** `15`
🔹 `*args` stores values in a **tuple**.

🧾 5. Keyword Variable-Length Arguments (kwargs)**

➡️ Use `**kwargs` when you want to pass **multiple keyword arguments** (key–value pairs).

```
def show_info(**data):
    for key, value in data.items():
        print(key, ":", value)

show_info(name="Atiar", age=21, dept="CSE")
```

✅ **Output:**

```
name : Atiar
age : 21
dept : CSE
```

🔹 `**kwargs` stores values in a **dictionary**.

🧩 Summary Table

Type	Symbol	Stores In	Example
Positional	—	Normal order	`add(2, 3)`
Keyword	—	Named pair	`add(b=3, a=2)`
Default	—	Parameter with default	`def f(a=5)`
Variable-length	`*args`	Tuple	`def f(*args)`
Keyword variable-length	`**kwargs`	Dictionary	`def f(**kwargs)`

What is recursion in python

✅ **Recursion in Python** means a **function calling itself** in order to solve a smaller version of the same problem.

It's a powerful concept that lets you write **clean and simple solutions** for problems that can be broken into sub problems — like factorial, Fibonacci, or tree traversal.

🧠 👉 Definition:

> **Recursion** is a process where a function **calls itself directly or indirectly** until a certain condition (base case) is met.

🧩 Basic Example

```python
def countdown(n):
    if n == 0:              # Base case (stopping condition)
        print("Done!")
    else:
        print(n)
        countdown(n - 1)    # Recursive call
```

Call:

```python
countdown(5)
```

Output:

```
5
4
3
2
1
Done!
```

⚙️ How Recursion Works

Every recursive function has **two parts**:

1. **Base Case** → when to stop recursion
2. **Recursive Case** → when the function calls itself again

📘 Example 2: Factorial Using Recursion

Mathematically:
[
n! = n × (n-1) × (n-2) × … × 1
]
or recursively:
[
n! = n × (n-1)!
]

```python
def factorial(n):
    if n == 1:              # Base case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive call

print(factorial(5))
```

✅ **Output:** `120`

⚠️ Important Points

- Every recursive function **must have a base case**, otherwise it will cause **infinite recursion** and a `RecursionError`.
- Python has a recursion limit (default ≈ 1000 calls).
  You can check it:

  ```python
  import sys
  print(sys.getrecursionlimit())
  ```

💡 Advantages

- Code looks **cleaner and shorter** for problems like factorial, Fibonacci, tree traversal.

❌ Disadvantages

- Can be **less efficient** (uses more memory and stack space).
- Harder to debug if base case is missing.

Let's clearly understand the **difference between recursion and iteration** in Python 👇

🔁 Recursion vs Iteration

Feature	Recursion	Iteration
Definition	A function **calls itself** repeatedly	A **loop** (`for` / `while`) repeats code

Feature	Recursion	Iteration
Control Mechanism	Function **calls itself** until base case	Loop runs until **condition** becomes false
Stopping Condition	**Base case** (explicit condition to stop)	**Loop condition** (e.g., `while n > 0`)
Memory Usage	Uses **stack memory** for each function call	Uses **constant memory**
Speed	**Slower** due to function call overhead	**Faster** (no extra function calls)
Readability	Shorter, elegant for recursive problems	More straightforward for repetitive tasks
Risk	May cause **RecursionError** if base case missing	May cause **infinite loop** if condition wrong
Examples	Factorial, Fibonacci, Tree Traversal	Counting, Summing, Repeated tasks

---

🧮 Example 1: Factorial using Recursion

```
def factorial_recursive(n):
    if n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)

print(factorial_recursive(5))
```

✅ **Output:** `120`

🔂 Example 2: Factorial using Iteration

```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iterative(5))
```

✅ **Output:** `120`

⚙️ Key Difference Summary

Aspect	Recursion	Iteration
Function Call	Yes	No
Memory Usage	More (stack)	Less
Termination	Base case	Loop condition
Best For	Divide & conquer problems	Simple repetitions

What is operator precedence

✅ **Operator Precedence** in Python means **the order in which operators are evaluated** when an expression has **multiple operators**.

It determines **which operation is done first**, just like in math (e.g., multiplication before addition).

🧠 👉 Definition:

> Operator precedence defines the **priority** of operators — which one Python executes **first** in a complex expression.

📘 Example

```
result = 10 + 5 * 2
print(result)
```

➡️ You might think it's `(10 + 5) * 2 = 30`,
but Python follows **precedence rules**, so it does `5 * 2` first →
✅ **Output:** `20`

Because **multiplication (`*`)** has **higher precedence** than **addition (`+`)**.

🧩 Python Operator Precedence (from highest to lowest)

Precedence Level	Operator(s)	Description / Example
🔝 1	`()`	Parentheses — highest priority
2	`**`	Exponentiation → `2 ** 3 = 8`
3	`+x`, `-x`, `~x`	Unary plus, minus, bitwise NOT
4	`*`, `/`, `//`, `%`	Multiplication, Division, Floor Div, Modulus
5	`+`, `-`	Addition, Subtraction
6	`<<`, `>>`	Bitwise shift operators
7	`&`	Bitwise AND
8	`^`	Bitwise XOR
9	`	`
10	`<`, `<=`, `>`, `>=`, `==`, `!=`	Comparison operators
11	`not`	Logical NOT
12	`and`	Logical AND
13	`or`	Logical OR
🔚 14	`=` `+=` `-=` etc.	Assignment operators (lowest precedence)

💡 Example Demonstrations

```
print(2 + 3 * 4)       # 3*4 first → 2 + 12 = 14
print((2 + 3) * 4)     # Parentheses first → 5 * 4 = 20
print(2 ** 3 ** 2)     # Right to left → 2 ** (3 ** 2) = 512
```

⚙️ Associativity

When operators have **same precedence**, Python uses **associativity** (left to right or right to left).

Operator Type	Associativity
`+`, `-`, `*`, `/`, `%`	Left → Right
`**`	Right → Left

Operator Type	Associativity
Assignment (`=` , `+=` , etc.)	Right → Left

🧮 Tip to Remember

Use **parentheses** `()` to make your code **clear and avoid confusion**:

```
result = (10 + 5) * 2   # Easier to read
```

---

what is the use "with" statement in python

🧠 What is the `with` Statement in Python?

➡️ The `with` **statement** in Python is used to **manage resources automatically** — like files, network connections, or database sessions — so you **don't have to close or clean them up manually**.

It simplifies code and helps avoid **resource leaks** (e.g., forgetting to close a file).

✅ 👉 Definition:

> The `with` statement simplifies the management of resources by **automatically handling setup and cleanup actions** using **context managers**.

📘 Basic Example (File Handling)

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

✅ What happens here:

1. `open("data.txt", "r")` → opens the file
2. `as file` → assigns it to the variable `file`

3. When the block finishes, Python **automatically closes** the file — even if an error occurs

No need to write:

```
file = open("data.txt", "r")
# do work
file.close()
```

⚙️ How It Works Internally

The `with` statement uses a **Context Manager**, which has two special methods:

- `__enter__()` → runs at the start
- `__exit__()` → runs when the block ends (even if there's an error)

Example (custom context manager):

```
class Demo:
    def __enter__(self):
        print("Entering block")
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting block")

with Demo():
    print("Inside block")
```

✅ **Output:**

```
Entering block
Inside block
Exiting block
```

💡 Common Use Cases

Use Case	Example
File Handling	`with open('file.txt') as f:`
Database Connection	`with db.connect() as conn:`
Thread Locking	`with lock:`
Temporary Files / Network Connections	Context managers clean up automatically

---

🎯 Benefits

✅ Automatically closes/cleans resources
✅ Cleaner, shorter code
✅ Prevents errors like forgetting `close()`
✅ Works even if exceptions occur

⚠️ Without `with`:

```
file = open("data.txt", "r")
content = file.read()
file.close()  # must remember to close manually
```

✅ With `with`:

```
with open("data.txt") as file:
    content = file.read()  # auto-closed after block
```

What is the difference between yeild and return

🧠 1. `return`

- Used inside a **function** to **send a value back to the caller** and **terminate the function** immediately.
- Returns a **single value** (or a tuple) and **exits** the function.
- Function is **not resumable** after return.

```
def get_numbers_return():
    numbers = [1, 2, 3]
    return numbers  # function ends here

print(get_numbers_return())
```

✅ **Output:** `[1, 2, 3]`

🧠 2. `yield`

- Used inside a **generator function** to **produce a value**, but **pause the function** instead of terminating it.
- Can **resume execution** from where it left off when the next value is requested.
- Returns a **generator object** instead of actual values at first.

```
def get_numbers_yield():
    for i in range(1, 4):
        yield i  # pauses here

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

✅ **Output:**

```
1
2
3
```

⚖️ Key Differences

Feature	`return`	`yield`
Purpose	Sends a value and **ends function**	Produces a value and **pauses function**
Function Type	Normal function	Generator function
Memory	Returns entire object at once	Returns **one value at a time** (memory-efficient)
Resumable?	❌ No	✅ Yes, can resume with `next()`
Use Case	When you need **all data at once**	When working with **large sequences / streams**

💡 Example: Return vs Yield

```python
def square_return(n):
    result = []
    for i in range(n):
        result.append(i*i)
    return result

def square_yield(n):
    for i in range(n):
        yield i*i

print(square_return(5))   # [0,1,4,9,16]

for val in square_yield(5):  # 0 1 4 9 16
    print(val, end=" ")
```

⚙️ When to Use Yield

- Large datasets (avoid memory overload)
- Infinite sequences (like `count()` or `Fibonacci`)
- Streaming data / pipelines

---

what is the difference between break, continue and pass

✅ Let's break down the difference between **`break`**, **`continue`**, and **`pass`** in Python clearly.

These are **control statements** used in loops (and sometimes conditionals), but they behave differently.

🧩 1. `break`

- **Purpose:** Exit the **entire loop** immediately.
- **Effect:** Stops the loop and moves execution to the next statement **after the loop**.

Example:

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

```python
        break  # exit the loop when i == 3
    print(i)
```

Output:

```
0
1
2
```

🧩 2. `continue`

- **Purpose: Skip the current iteration** of the loop and continue with the **next iteration**.
- **Effect:** Loop doesn't stop, just jumps to the next cycle.

Example:

```python
for i in range(5):
    if i == 3:
        continue  # skip when i == 3
    print(i)
```

Output:

```
0
1
2
4
```

🧩 3. `pass`

- **Purpose:** Do **nothing**. It's a **placeholder**.
- **Effect:** No operation is performed, but **syntax is valid**.

Example:

```python
for i in range(5):
    if i == 3:
        pass  # do nothing
    print(i)
```

Output:

```
0
1
2
3
4
```

✅ Often used when:

- Writing **empty functions or classes**
- Placeholder for code to be implemented later
- Keeping the syntax correct

⚖️ Comparison Table

Statement	Action	Stops Loop?	Skips Iteration?	Placeholder?
`break`	Exit the loop entirely	✅ Yes	❌ No	❌ No
`continue`	Skip current iteration	❌ No	✅ Yes	❌ No
`pass`	Do nothing	❌ No	❌ No	✅ Yes

💡 Analogy

- **`break`** → Stop the ride immediately.
- **`continue`** → Skip this step, keep going.
- **`pass`** → Stand still for now, do nothing.

Monkey Patching clearly. ✅

🧠 What is Monkey Patching in Python?

➡️ **Monkey patching** is the technique of **modifying or extending a class or module at runtime**, without changing the original source code.

It's like **dynamically "patching" a class or function** while your program is running.

✅ Definition

> Monkey patching allows you to **change the behavior of functions, methods, or classes on-the-fly** at runtime, often for testing or quick fixes.

📘 Basic Example

```python
class Person:
    def greet(self):
        print("Hello!")

# Original behavior
p = Person()
p.greet()  # Output: Hello!

# Monkey patching: change greet method dynamically
def new_greet(self):
    print("Hi, I am monkey patched!")

Person.greet = new_greet

p.greet()  # Output: Hi, I am monkey patched!
```

✅ Notice that we didn't change the **original class code**, but its behavior changed at runtime.

⚙️ Common Use Cases

1. **Fixing bugs in third-party libraries** without editing the source.
2. **Testing**: Replace functions/methods with mocks or stubs.
3. **Extending functionality** dynamically.

⚠️ Cautions

- Can make code **hard to read and debug**.
- Overusing it may lead to **unexpected behavior**, especially in large projects.
- Best used **carefully**, mainly in testing or temporary fixes.

---

pickling and `unpickling` in Python

🧠 1. What is Pickling?

➡️ **Pickling** is the process of **converting a Python object into a byte stream** (binary format) so it can be **saved to a file, sent over a network, or stored**.

- This allows Python objects to be **serialized** and stored or transmitted.
- The module used is **`pickle`**.

Example: Pickling

```
import pickle

data = {"name": "Atiar", "age": 21}

# Save data to a file
with open("data.pkl", "wb") as file:
    pickle.dump(data, file)  # pickling
```

✅ Here:

- `wb` → write binary mode
- `pickle.dump()` → converts Python object into bytes and writes it to a file

---

🧠 2. What is Unpickling?

➡️ **Unpickling** is the process of **converting the byte stream back into the original Python object**.

- The object is restored to its original form.

Example: Unpickling

```
import pickle

# Load data from file
with open("data.pkl", "rb") as file:
    loaded_data = pickle.load(file)  # unpickling

print(loaded_data)
```

✅ Output:

```
{'name': 'Atiar', 'age': 21}
```

- `rb` → read binary mode
- `pickle.load()` → converts byte stream back to Python object

⚙️ Key Points

Feature	Pickling	Unpickling
Purpose	Serialize Python object	Deserialize Python object
Function	`pickle.dump(obj, file)`	`pickle.load(file)`
Output	Byte stream	Python object
File Mode	`wb`	`rb`

💡 Use Cases

- Saving **model objects** in machine learning
- Storing **data structures** for later use
- Sending Python objects over **network**

⚠️ Cautions

- Pickle files can **execute arbitrary code** → don't unpickle files from untrusted sources.
- Only works for **Python objects** (not all external objects like open file handles).

---

how can i handle memory leak in python

🧠 What is a Memory Leak?

➡️ A **memory leak** happens when **memory is allocated but not released**, causing your program to **use more and more memory over time**.

Even though Python has **automatic garbage collection**, memory leaks can still occur due to:

- Circular references that aren't properly cleaned
- Global or long-lived references holding large objects
- Third-party libraries mismanaging memory

🛠 How to Handle Memory Leaks in Python

1️⃣ Use Proper Scope

- Avoid storing **large objects in global variables**.
- Limit object lifetime by keeping variables **local whenever possible**.

```
def process_data():
    data = [i for i in range(1000000)]  # local variable
    # process data
```

2️⃣ Delete Unused Objects

- Use `del` to remove variables or objects no longer needed.

```
data = [i for i in range(1000000)]
# process data
del data  # free memory
```

3️⃣ Garbage Collection (gc module)

- Python uses **reference counting** and **garbage collection**.
- You can **manually check and clean memory** using the `gc` module.

```
import gc

gc.collect()  # force garbage collection
```

- Use `gc.get_objects()` to inspect current objects.

4️⃣ Avoid Circular References

- Circular references occur when objects reference each other.
- Python's GC usually handles them, but sometimes objects with `__del__` can cause leaks.

```
class Node:
    def __init__(self):
        self.ref = None

a = Node()
b = Node()
a.ref = b
b.ref = a  # circular reference

del a
del b
import gc
gc.collect()  # ensures circular references are cleaned
```

5️⃣ Use Generators Instead of Lists

- For large datasets, use **generators** (`yield`) instead of storing everything in memory.

```
def large_data():
    for i in range(1000000):
        yield i  # yields one item at a time
```

6️⃣ Profile Memory Usage

- Use **memory profiling tools** to detect leaks:
  - `tracemalloc` — tracks memory allocation
  - `memory_profiler` — line-by-line memory usage

```
import tracemalloc

tracemalloc.start()
# run code
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')
```

```
for stat in top_stats[:10]:
    print(stat)
```

7️⃣ Be Careful With Third-Party Libraries

- Some C-extensions or libraries may **leak memory** if objects are not properly released.
- Always check library docs for **proper cleanup methods**.

💡 Summary

- Keep **object lifetimes short**
- Use **del** and **gc.collect()** when needed
- Prefer **generators** for large data
- Profile memory regularly
- Avoid **unnecessary circular references**

---

python closure

🧠 What is a Closure in Python?

➡️ A **closure** is a **function object that remembers values from its enclosing lexical scope**, even if the outer function has finished execution.

In simple words:

> A closure allows a **nested function** to **remember and access variables from its parent function** even after the parent function has returned.

🔑 Conditions for a Closure

1. There must be a **nested (inner) function**.
2. The inner function must **refer to a variable from the outer function**.
3. The outer function must **return the inner function**.

📘 Basic Example

```python
def outer(x):
    def inner(y):
        return x + y  # inner uses x from outer
    return inner     # return the inner function

closure_func = outer(10)  # outer() finished, but x=10 is remembered
print(closure_func(5))    # 15
```

✅ Here:

- `inner()` is a **closure** that remembers `x` from `outer()`
- Even though `outer()` has finished, `inner()` still **remembers** **`x=10`**

🔍 Why Use Closures?

1. **Data Hiding / Encapsulation**
    - Variables from the outer function are **not accessible globally**
2. **Maintain State Between Calls**
    - Useful for **counters, accumulators, or configuration**
3. **Avoid Global Variables**
    - Helps write cleaner code

📘 Example: Counter Using Closure

```python
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c = counter()
print(c())  # 1
print(c())  # 2
print(c())  # 3
```

✅ Here:

- `increment()` remembers `count` even though `counter()` has finished
- `nonlocal` is used to modify the outer variable

⚡ Key Points

- A closure **remembers values from its enclosing scope**
- Useful for **stateful functions without using classes**
- Often used in **decorators, callbacks, and functional programming**

💡 Analogy

Think of a closure like a **backpack**:

- Outer function packs some items (`x`)
- Inner function carries the backpack wherever it goes, even after the outer function ends

---