Difference between python 2 and python 3

Key Differences

Feature	Python 2	Python 3
Release Year	2000	2008
Support	End of life in 2020 (no updates)	Actively supported
Print Statement	print "Hello" (statement)	print("Hello") (function)
Division	5/2 = 2 (integer division by default)	5/2 = 2.5 (true division by default)
Unicode	Text is ASCII by default, use u"Hello" for Unicode	Strings are Unicode by default ("Hello")
xrange / range	range() returns a list, xrange() is for iteration	Only range(), works like xrange() (lazy sequence)
Iterators	.next() method used	next() function used
Error Handling	except Exception, e:	except Exception as e:
Input	raw_input() for strings, input() evaluates expression	input() always returns string
Integer Types	Separate int and long types	Only int (no limit, arbitrary precision)
Libraries	Many old libraries only worked with Python 2	Modern libraries support Python 3
Community	Deprecated	Fully active

Python 2 → Legacy, outdated, no official support.
Python 3 → Future-proof, cleaner syntax, Unicode by default, better division, more consistent.

- **what is indentation in python and why is it important?**
  Indentation means adding spaces or tabs at the beginning of a line.In many languages (like Java, C, C++), indentation is just for readability, and the compiler ignores it.But in Python, indentation is mandatory and defines the code blocks (like loops, functions, conditionals, and classes).

Why is it important?

1. Defines Code Blocks :- In Python, there are no {} braces or end keywords.Indentation tells Python where a block starts and ends.

Example:

```
if True:
   print("Inside block")
   print("Still inside")
print("Outside block")
```

2. Avoids Ambiguity :- Forces developers to write clean, structured, and readable code.
3. Readability & Consistency:- Python follows the philosophy of being clean and human-readable.“There should be one—and preferably only one—obvious way to do it.” (Zen of Python).

Interview-Safe Answer

“Indentation in Python refers to the spaces at the beginning of a line that define code blocks. Unlike other languages that use curly braces, Python uses indentation to determine the grouping of statements in loops, conditionals, and functions. It is important because without proper indentation, Python code will throw errors. This enforces readability and consistency across Python codebases.”

- **Mutable vs immutable types (examples)**

“Mutable objects in Python can be modified after creation, like lists, dictionaries, and sets. Immutable objects cannot be changed after creation—examples include strings, tuples, and numbers. With immutables, any modification creates a new object in memory, whereas mutables change in place.”

```
# Mutable
a = [1, 2, 3]
print(id(a))      # e.g. 140352
a.append(4)
print(id(a))      # same id (changed in place)
# Immutable
b = "hello"
print(id(b))      # e.g. 150120
b = b + " world"
print(id(b))      # different id (new object created)
```

140347612345200
140347612347600
First print → the original string "hello" has some memory address (e.g., 140347612345200).
Second print → after concatenation, a new string "hello world" is created at a different memory address (e.g., 140347612347600).
This proves that strings are immutable—a new object is created instead of modifying the old one.

- Explain **python variable and scope (LEGB) Rule** Python Variable

A variable is a name that refers to a value/object stored in memory.
Example:- x = 10 # x is a variable referring to the integer object 10 name = "Swapnil" Variable Scope.
Scope is the region of a program where a variable can be accessed. Python uses the LEGB rule to determine the order in which it looks for a variable:

Letter	Scope Type	Description	Example
L	Local	Inside the current function	def func(): x = 5 → x is local
E	Enclosing	Variables in enclosing function(s) (non-local)	Nested functions: def outer(): y = 10; def inner(): print(y)
G	Global	Module-level variables	x = 100; def func(): print(x)
B	Built-in	Python reserved names/functions	len(), print(), int()

what is type casting in python?
“Type casting in Python is the process of converting a variable from one data type to another. It can be implicit (automatic conversion by Python) or explicit (manual conversion using functions like int(), float(), str(), list(), etc.). Implicit type casting usually happens with numeric types, while explicit casting allows you to safely convert between compatible types.”
Implicit → automatic, safe, usually numeric types.
Explicit → manual, using type conversion functions.
Cannot convert incompatible types (e.g., int("abc") will raise ValueError).
Type Casting in Python :- Type casting means converting a value from one data type to another.Sometimes called type conversion.
Why do we need it? :- To perform operations between different data types.To ensure correct input/output type.To avoid errors in calculations or function calls.

Types of Type Casting

1. Implicit Type Casting (Type Conversion) :- Python automatically converts one data type to another without user intervention.Usually happens with numeric types.

Example:

```
a = 5       # int
b = 2.0     # float
c = a + b   # Python converts a → float
print(c)    # 7.0
print(type(c))  # <class 'float'>
```

2. Explicit Type Casting :- User manually converts one type to another using built-in functions.Common functions: int(), float(), str(), list(), tuple(), set(), etc.

Examples:

```
# int to float
x = 10
y = float(x)
print(y, type(y))  # 10.0 <class 'float'>

# float to int
a = 9.8
b = int(a)
print(b, type(b))  # 9 <class 'int'>

# int to string
num = 100
s = str(num)
print(s, type(s))  # '100' <class 'str'>

# string to list
text = "hello"
lst = list(text)
print(lst)  # ['h', 'e', 'l', 'l', 'o']
```

Difference between List, Tuple, Set and Dictionary

Feature	List	Tuple	Set	Dictionary
Definition	Ordered, mutable sequence of items	Ordered, immutable sequence of items	Unordered, mutable collection of unique items	Unordered, mutable collection of key-value pairs
Syntax	[1, 2, 3]	(1, 2, 3)	{1, 2, 3}	{"a": 1, "b": 2}
Mutability	Mutable → can add, remove, update	Immutable → cannot change after creation	Mutable → can add/remove elements, but no duplicates	Mutable → can add/update/remove key-value pairs
Duplicates	Allowed	Allowed	Not allowed	Keys unique, values can be duplicated
Indexing / Slicing	✅ Yes	✅ Yes	❌ No (unordered)	❌ No, but access via keys

Feature	List	Tuple	Set	Dictionary
Definition	Ordered, mutable sequence of items	Ordered, immutable sequence of items	Unordered, mutable collection of unique items	Unordered, mutable collection of key-value pairs
Syntax	[1, 2, 3]	(1, 2, 3)	{1, 2, 3}	{"a": 1, "b": 2}
Mutability	Mutable → can add, remove, update	Immutable → cannot change after creation	Mutable → can add/remove elements, but no duplicates	Mutable → can add/update/remove key-value pairs
Use Case / Interview Tip	Use when order matters and you may need to change elements	Use for fixed data, faster than list, can be dictionary keys	Use for unique elements, membership testing fast	Use for key-value mapping, lookups by key

1️⃣ List (mutable & ordered)

```
my_list = [1, 2, 3, 4]
my_list.append(5)      # [1, 2, 3, 4, 5]
my_list[0] = 100       # [100, 2, 3, 4, 5]
```

2️⃣ Tuple (immutable & ordered)

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 100   # ❌ Error
```

3️⃣ Set (unordered, unique)

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

4️⃣ Dictionary (key-value mapping)

```
my_dict = {"a": 1, "b": 2}
my_dict["c"] = 3       # {"a": 1, "b": 2, "c": 3}
my_dict["a"] = 100     # {"a": 100, "b": 2, "c": 3}
```

In Python, a list is an ordered and mutable collection; a tuple is ordered but immutable. A set is an unordered collection of unique elements, useful for membership testing, and a dictionary stores key-value pairs for fast lookups. Lists and tuples maintain order, sets remove duplicates and are unordered, and dictionaries allow mapping from unique keys to values."

when to use list vs tuple?

Aspect	List	Tuple
Mutability	Mutable → can change, add, remove elements	Immutable → cannot change after creation
Use Case	When you need a dynamic collection of items that may change over time	When you need a fixed collection of items that shouldn't change
Performance	Slower than tuple because of mutability overhead	Faster than list → lightweight, optimized for iteration
Memory	Consumes more memory due to dynamic nature	Consumes less memory
Hashability	Cannot be used as dictionary keys (unless frozen)	Can be used as dictionary keys (if all elements are immutable)
Methods	Many built-in methods like append(), remove(), pop()	Fewer methods (mainly count, index)

Example 1: List

```
tasks = ["email", "meeting", "call"]
tasks.append("lunch")
print(tasks)
```
Output:
```
['email', 'meeting', 'call', 'lunch']
```
✅ The list was modified in place.

Example 2: Tuple
```
coordinates = (10.0, 20.0)
print(coordinates)
# coordinates[0] = 15.0  # Uncommenting this will cause an error
```
Output:
```
(10.0, 20.0)
```
If you try to modify the tuple:
```
coordinates[0] = 15.0
```
Error:
```
TypeError: 'tuple' object does not support item assignment
```
This clearly shows:

- List → mutable → can change
- Tuple → immutable → cannot change

How to remove duplicates from the list ?

1️⃣Using set()
```
my_list = [1, 2, 2, 3, 4, 4, 5]
unique_list = list(set(my_list))
print(unique_list)  # Output: [1, 2, 3, 4, 5] (order may change)
```
2️⃣Using dict.fromkeys() (preserves order)
```
my_list = [1, 2, 2, 3, 4, 4, 5]
unique_list = list(dict.fromkeys(my_list))
print(unique_list)  # Output: [1, 2, 3, 4, 5]
```

- Use set() → fast, no order needed.
- Use dict.fromkeys() or loop with set → preserves order.
- For speed and if order doesn't matter: Use set().
- For preserving order and still being efficient: Use dict.fromkeys() or the list comprehension with a set.
- For explicit control or if elements are unhashable: Use the loop with a temporary list.
-

what is dictionary comprehension?

- Dictionary comprehension is a compact way to create dictionaries in Python using a single line, similar to list comprehension.
  Syntax:- {key_expr: value_expr for item in iterable if condition}

Dictionary comprehension is a concise and efficient Python feature for creating dictionaries from iterables in a single line, similar to list comprehensions but with curly braces {} and a key:value structure. It allows you to generate new key-value pairs by transforming elements from an existing iterable, optionally filtering them with an if condition, thereby replacing verbose for loops and if statements.

1️⃣Basic Example
```
# Create a dictionary of squares
squares = {x: x**2 for x in range(5)}
print(squares)
```
Output:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```
2️⃣With Condition
```
# Only even numbers
```

```
even_squares = {x: x**2 for x in range(5) if x % 2 == 0}
print(even_squares)
```
Output:{0: 0, 2: 4, 4: 16}

“Dictionary comprehension in Python is a concise way to create dictionaries using a single line of code, similar to list comprehension. It allows you to define key-value pairs for each item in an iterable, optionally with a condition.”

- If asked “difference between list comprehension and dict comprehension”, you can say:

“List comprehension produces a list of values, while dictionary comprehension produces key-value pairs in a dictionary. The main visual cue is the curly braces with a colon for dict: {key: value} vs brackets [value] for list.”

What is frozen set in python ?

- A frozenset is an immutable version of a set.Once created, you cannot add, remove, or modify elements.Useful when you need a set that should not change or want to use it as a dictionary key or element of another set
- Key Features

1. Immutable → cannot change after creation.
2. Supports all set operations like union, intersection, difference.
3. Can be used as a key in a dictionary (normal sets cannot).

🔹 Examples

1️⃣ Creating a frozenset
```
my_set = frozenset([1, 2, 3, 2])
print(my_set)
```
Output:- frozenset({1, 2, 3})

2️⃣ Trying to modify it
```
my_set.add(4)  # ❌ AttributeError: 'frozenset' object has no attribute 'add'
```
3️⃣ Using frozenset as dictionary key
```
d = {frozenset([1,2,3]): "value"}
print(d)
```
Output: - {frozenset({1, 2, 3}): 'value'}

When to Use

- **When you need a set that should not change.**

“A frozenset in Python is an immutable version of a set. It cannot be modified after creation but supports set operations like union and intersection. It is hashable, so it can be used as a dictionary key or as an element of another set.”

How is string slicing done in python?

String Slicing in Python

- Slicing lets you extract a substring from a string (or elements from lists, tuples, etc.) using indices.
- Syntax:

```
string[start:stop:step]
```

Parameter	Description
start	Starting index (inclusive). Defaults to 0 if omitted.
stop	Ending index (exclusive). Slice goes up to stop-1.
step	Step size (interval between elements). Defaults to 1.

```
s = "Python"
# Basic slicing
print(s[0:4])   # 'Pyth'  → from index 0 to 3
# Omit start (from beginning)
print(s[:3])    # 'Pyt'
# Omit stop (till end)
print(s[2:])    # 'thon'
# Negative indices
print(s[-4:-1]) # 'tho' → counts from end
```

```
# Step
print(s[0:6:2]) # 'Pto' → every 2nd character
# Reverse string
print(s[::-1])  # 'nohtyP'
```

1. start is inclusive, stop is exclusive.
2. Negative indices count from the end (-1 is last character).
3. step can be negative → reverses the string or sequence.

Interview-Safe Answer
“In Python, string slicing allows you to extract a portion of a string using the syntax [start:stop:step]. The start index is inclusive, the stop index is exclusive, and the step defines the interval between characters. You can also use negative indices to slice from the end or use a negative step to reverse the string.”

Difference between break , continue , pass with examples
“break is used to exit a loop immediately, continue is used to skip the current iteration and move to the next, and pass is a placeholder that does nothing. break and continue control loop flow, while pass is often used when code is syntactically required but not implemented yet.”
ˇ

Keyword	Action	Typical Use	Example
break	Exit loop immediately	Stop looping when condition met	Exit a search when item found
continue	Skip current iteration	Ignore certain cases but continue loop	Skip even numbers while printing
pass	Do nothing	Placeholder for future code	Empty function, class, loop

1. break in Python
Theory

- break is used to exit the nearest enclosing loop immediately, regardless of the loop condition.
- Commonly used when a condition is met and you want to stop looping.

Example

```
for i in range(1, 6):
   if i == 3:
      break
   print(i)
```

Output:
1
2
Explanation:

- When i == 3, the break statement executes.
- Loop terminates immediately; code after break inside the loop is not executed.

Interview Perspective
“Use break when you want to terminate a loop prematurely based on a condition.”

2. continue in Python
Theory

- continue skips the rest of the current iteration of the loop and moves to the next iteration.
- Useful when you want to ignore certain cases but continue looping.

Example

```
for i in range(1, 6):
   if i == 3:
      continue
```

```
    print(i)
```

Output:
1
2
4
5
Explanation:

- When i == 3, continue executes → skips the print statement for 3.
- Loop continues with i = 4.

Interview Perspective
“Use continue when you want to skip certain iterations but not exit the loop entirely.”

3. pass in Python
Theory

- pass does nothing; it’s a placeholder.
- Often used when a statement is syntactically required but no action is needed.
- Common use cases: empty functions, loops, or classes during development.

Example

```
for i in range(1, 6):
    if i == 3:
        pass
    print(i)
```

Output:
1
2
3
4
5

Explanation:

- pass does nothing → loop continues normally.
- Useful as a placeholder for code to be implemented later.

Another Example (empty function):

```
def my_function():
    pass
```

- This is valid Python even though the function body is empty.

Interview Perspective
“Use pass as a placeholder when a statement is syntactically required but no action is needed. It is often used in function or class definitions during development.”

For vs While loop
“for loop is used when you know the number of iterations or need to iterate over a sequence. while loop is used when you don’t know the exact number of iterations and want to repeat a block until a condition becomes false.”
1. for loop
Theory

- Used to iterate over a sequence (like list, tuple, string, or range).
- Number of iterations is known or finite.

Syntax

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

Example 1: Iterating over a list

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

Output:
apple
banana
cherry

Example 2: Using range()

```
for i in range(1, 6):
    print(i)
```

Output:
1
2
3
4
5

When to use

- When you know how many times to loop or need to iterate over items.

2. while loop

Theory

- Repeats a block of code while a condition is true.
- Number of iterations is not necessarily known.

Syntax

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

Example 1: Basic while loop

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

Output:
1
2
3
4
5

Example 2: Using user input

```
password = ""
while password != "secret":
    password = input("Enter password: ")
print("Access granted")
```

When to use

- When you don't know how many times the loop will run, but need to loop until a condition changes.

Feature	for loop	while loop
Iteration	Over sequence or range	Based on a condition
Known iterations	Yes	No (could be infinite)

Use case	Iterate list, string, range	Repeat until condition met
Syntax	for item in iterable:	while condition:

“for loop is used when you know the number of iterations or need to iterate over a sequence. while loop is used when you don’t know the exact number of iterations and want to repeat a block until a condition becomes false.”

how to use enumerate in for loop in python

The enumerate() function in Python is used within a for loop to iterate over an iterable (like a list, tuple, or string) while simultaneously getting both the index and the value of each item. This avoids the need to manually manage a counter variable.
“enumerate() in Python is used to loop over an iterable while keeping track of the index along with the value. It returns pairs of (index, value), and we can also specify a starting index. It is more Pythonic and readable than manually using range(len()).”

enumerate() adds a counter/index to an iterable (like list, tuple, or string).
It returns an enumerate object (which is iterable of pairs: (index, item)).
Often used in for loops when you need both index and value.
Syntax
enumerate(iterable, start=0)

- iterable → list, tuple, string, etc.
- start → index to start counting from (default = 0).

Basic Example

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

for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:
0 apple
1 banana
2 cherry

Start index from 1

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

Output:
1 apple
2 banana
3 cherry

With a string

```
for i, ch in enumerate("Python"):
    print(i, ch)
```

Output:
0 P
1 y
2 t

3 h
4 o
5 n
Cleaner and more Pythonic than using range(len(iterable)).Improves readability in loops. Avoids manually managing counters.
range(len(iterable)) → more manual, less readable.
enumerate(iterable) → cleaner, Pythonic, gives index and value in one step.
enumerate(iterable, start=1) → useful for human-readable numbering.

what is zip() in python?
zip() in Python is used to combine multiple iterables element-wise. It returns an iterator of tuples, where each tuple contains elements from the input iterables at the same position. If the iterables are of different lengths, zip() stops at the shortest one. It's commonly used for parallel iteration, creating dictionaries, or grouping data together."
The zip() function in Python combines multiple iterables such as lists, tuples, strings, dict etc, into a single iterator of tuples. Each tuple contains elements from the input iterables that are at the same position.
Let's consider an example where we need to pair student names with their test scores:

```
names = ['John', 'Alice', 'Bob', 'Lucy']
scores = [85, 90, 78, 92]
res = zip(names, scores)
print(list(res))
```

Output
[('John', 85), ('Alice', 90), ('Bob', 78), ('Lucy', 92)]
Explanation:

- zip() is used to combine the two lists into a single iterable 'res'
- Each element from names is paired with the corresponding element from scores
- list() converts the iterator from zip() into a list of tuples, making it easier to visualize or manipulate the combined data.

Iterables of different Lengths
When using iterables of different lengths, the zip() will only pair up to the shortest iterable.

```
names = ['Alice', 'Bob', 'Charlie']
scores = [88, 94]
res = zip(names, scores)
print(list(res))
```

Output
[('Alice', 88), ('Bob', 94)]

Explanation: Here, zip() stops after pairing the two available score values with the first two name values. 'Charlie' is left out since there's no corresponding score value.
Unzipping data with zip()
We can also reverse the operation by unzipping the data using the * operator. Let's see how that works:

```
a = [('Apple', 10), ('Banana', 20), ('Orange', 30)]
fruits, quantities = zip(*a)
print(f"Fruits: {fruits}")
print(f"Quantities: {quantities}")
```

Output
Fruits: ('Apple', 'Banana', 'Orange')
Quantities: (10, 20, 30)
Explanation: Using the * operator, we can separates (unzip) the paired fruit names and their quantities back into their respective sequences

How to define a function in python?
Defining a Function in Python

In Python, a function is defined using the def keyword.
Functions allow you to reuse code, organize logic, and make your program modular.
✅ Syntax

```
def function_name(parameters):
    """
    Optional docstring (description of the function)
    """
    # function body
    return value
```

- def → keyword to define a function
- function_name → identifier (name of the function)
- parameters → (optional) values passed into the function
- return → (optional) value sent back to the caller

Python Functions are a block of statements that does a specific task. The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Different Between *args and **kwargs?
Both *args and **kwargs are used in function definitions to handle variable-length arguments (when you don't know how many arguments will be passed).

Feature	***args**	****kwargs**
Meaning	Arbitrary positional arguments	Arbitrary keyword arguments
Data Structure	Tuple	Dictionary
Usage	Variable number of values	Variable number of key-value pairs
Example Call	func(1, 2, 3)	func(x=1, y=2)

Use *args when you want to handle extra positional arguments.
Use **kwargs when you want to handle extra named arguments.
*args → Non-keyword Variable Arguments

- Collects positional arguments into a tuple.
- Useful when you don't know how many positional arguments will be passed.
- *args allows a function to accept an arbitrary number of non-keyworded, or positional, arguments.

Example:

```
def add_numbers(*args):
    print("Arguments:", args)      # args is a tuple
    return sum(args)
print(add_numbers(2, 3, 5))
print(add_numbers(1, 2, 3, 4, 5))
```

🔹 **Output**:

```
Arguments: (2, 3, 5)
10
Arguments: (1, 2, 3, 4, 5)
15
```

Here, *args packed values (2, 3, 5) into a tuple.

****kwargs → Keyword Variable Arguments**

- Collects keyword arguments into a dictionary.
- Useful when you don't know how many keyword arguments will be passed.
- **kwargs allows a function to accept an arbitrary number of keyworded, or named, arguments.

- The ** before kwargs signifies that all keyword arguments passed to the function will be collected into a dictionary named kwargs.

Example:

```
def print_user_info(**kwargs):
    print("Keyword Arguments:", kwargs)   # kwargs is a dictionary
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_user_info(name="Swapnil", age=30, city="Pune")
```

Output:

```
Keyword Arguments: {'name': 'Swapnil', 'age': 30, 'city': 'Pune'}
name: Swapnil
age: 30
city: Pune
```

Here, **kwargs packed arguments into a dictionary.

You can use both *args and **kwargs in the same function, but order matters:

def func(positional, *args, keyword_only, **kwargs)

Example:

```
def demo_function(a, b, *args, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)        # tuple
    print("kwargs:", kwargs)    # dict

demo_function(1, 2, 3, 4, 5, x=10, y=20)
```

🔹 **Output**:

```
a: 1
b: 2
args: (3, 4, 5)
kwargs: {'x': 10, 'y': 20}
```

What is Recursion?

👉 Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem.

- Each recursive call should bring the problem closer to a base case (the condition where the recursion stops).
- Without a base case, recursion would run infinitely and cause a RecursionError.

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

Recursion is when a function calls itself.

Requires a base case to avoid infinite calls.

Recursion is often used for problems that can be broken into smaller subproblems (e.g., factorial, Fibonacci, tree/graph traversal, quicksort, mergesort).

Recursion can sometimes be less efficient than iteration (because of extra function calls and stack memory usage).

Python has a recursion depth limit (default ~1000).

```
def factorial(n):
    if n == 0:   # base case
        return 1
    else:
        return n * factorial(n - 1)   # recursive call
print(factorial(5))   # 5*4*3*2*1 = 120
```

🔹 Output:

```
120
```

What is a Lambda Function?

A lambda function is a small, anonymous function in Python, defined using the keyword lambda.A lambda function in Python is a small, anonymous function defined with the lambdakeyword. Unlike standard functions defined with def, lambda functions are restricted to a single expression, which is implicitly returned. They do not require a name, hence they are often referred to as "anonymous functions." syntax:-lambda arguments: expression

It can take any number of arguments but must have only one expression (evaluated and returned).

- Syntax:

lambda arguments: expression

Example:

```
square = lambda x: x * x
print(square(5))   # 25
```

1. Short, throwaway functions
   - When you need a function only once, without formally defining it using def.
     Example:

```
print((lambda a, b: a + b)(5, 10))  # 15
```

Commonly Used Python Built-in Functions

Here are some of the most frequently used built-in functions (with quick examples):

✅ 1. len() → Get length

```
s = "python"
print(len(s))   # 6
```

✅ 2. type() → Check type

```
print(type(123))      # <class 'int'>
print(type([1, 2]))   # <class 'list'>
```

✅ 3. id() → Memory address

```
x = 10
print(id(x))
```

✅ 4. max(), min() → Find max/min

```
nums = [3, 7, 1, 9]
print(max(nums))  # 9
print(min(nums))  # 1
```

✅ 5. sum() → Sum of iterable

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

✅ 6. sorted() → Return sorted list

```
nums = [5, 2, 8]
print(sorted(nums))         # [2, 5, 8]
print(sorted(nums, reverse=True))  # [8, 5, 2]
```

✅ 7. zip() → Combine iterables

```
names = ["A", "B", "C"]
scores = [90, 80, 70]
print(list(zip(names, scores)))
# [('A', 90), ('B', 80), ('C', 70)]
```

✅ 8. enumerate() → Index with items

```
fruits = ["apple", "banana"]
for i, fruit in enumerate(fruits):
    print(i, fruit)
```

Output:

```
0 apple
1 banana
```

✅ 9. map() → Apply function

```
nums = [1, 2, 3]
```

```
squares = list(map(lambda x: x*x, nums))
print(squares)  # [1, 4, 9]
✅ 10. filter() → Filter values
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)  # [2, 4]
```

Class and object in Python

A class is a blueprint or template for creating objects.

- It defines attributes (variables/data) and methods (functions/behavior).
- Doesn't hold actual data itself — only describes how objects should look/behave.

Class

A class is a blueprint or a template for creating objects. It defines a set of attributes (variables) and methods (functions) that the objects created from it will possess.Classes encapsulate data and behavior into a single unit, promoting code organization and reusability. In essence, a class describes what an object of that type will be and what it can do.

An object is an instance of a class. It is a concrete realization of the blueprint defined by the class. Each object created from a class will have its own distinct set of attribute values, while sharing the same methods defined in the class. Objects are the entities through which you interact with the data and functionality defined in a class

```
class Dog:
  # Class attribute
  species = "Canis familiaris"

  # Initializer / Constructor method
  def __init__(self, name, age):
    self.name = name  # Instance attribute
    self.age = age    # Instance attribute

  # Instance method
  def bark(self):
    return f"{self.name} says Woof!"

# Creating objects (instances) of the Dog class
my_dog = Dog("Buddy", 3)
your_dog = Dog("Lucy", 5)

# Accessing attributes and calling methods through objects
print(f"{my_dog.name} is a {my_dog.species} and is {my_dog.age} years old.")
print(my_dog.bark())

print(f"{your_dog.name} is a {your_dog.species} and is {your_dog.age} years old.")
print(your_dog.bark())
```

In this example:

- Dog is the class.
- species is a class attribute, shared by all Dog objects.
- __init__ is the constructor method, used to initialize instance attributes when an object is created.
- name and age are instance attributes, unique to each Dog object.
- bark is an instance method, defining an action a Dog object can perform.
- my_dog and your_dog are objects (instances) of the Dog class. They each have their own name and age, but share the species attribute and the barkmethod.

1. Instance Method
- The most common method type.
- Takes self as the first argument (represents the object instance).
- Can access & modify instance variables and class variables.

Example:

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

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

s1 = Student("Swapnil", 95)
print(s1.show())   # Swapnil scored 95 marks
```

2. Class Method (@classmethod)
- Declared using the @classmethod decorator.
- Takes cls as the first argument (represents the class, not the object).
- Can access and modify class variables, but not instance variables.

Example:

```
class Student:
    school_name = "ABC School"   # class variable

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

    @classmethod
    def get_school(cls):   # Class method
        return f"School: {cls.school_name}"

print(Student.get_school())   # School: ABC School
```

3. Static Method (@staticmethod)
- Declared using the @staticmethod decorator.
- Does not take self or cls as the first parameter.
- Cannot access or modify instance variables or class variables directly.
- Used for utility/helper functions that logically belong to the class.

Example:

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

print(MathUtils.add(5, 7))   # 12
```

How do you import a module in python?

In Python, modules are imported using the import statement. There are several ways to import a module or its components: Import the entire module.This is the most common way to import a module. It makes all the functions, classes, and variables defined within the module accessible by prefixing them with the module name. Import specific items from a module.

```
import math
  print(math.pi)
  print(math.sqrt(25))
```

2. Import with alias (nickname)
import math as m
print(m.sqrt(25)) # 5.0
3. Import specific functions / classes
from math import sqrt, pi
print(sqrt(9)) # 3.0
print(pi) # 3.141592653589793

👉 No need to prefix with math..

1. import module:
- This statement imports the entire module and makes it available as a module object in the current namespace.
- To access any function, class, or variable within the module, you must prefix it with the module name.

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

2. from module import name:
- This statement imports specific names (functions, classes, or variables) directly into the current namespace.
- You can then use these imported names directly without needing to prefix them with the module name.

```
  from math import pi, sqrt
  print(pi)
  print(sqrt(25))
```

One of the features of Python is that it allows users to organize their code into modules and packages, which are collections of modules. The __init__.py file is a Python file that is executed when a package is imported. In this article, we will see what is __init__.py file in Python and how it is used in Python.

What Is __Init__.Py File in Python?

The __init__.py file is a Python file that is executed when a package is imported. __init__.py is a special file used in Python to define packages and initialize their namespaces. It can contain an initialization code that runs when the package is imported. Without this file, Python won't recognize a directory as a package. It serves two main purposes:

It marks the directory as a Python Package so that the interpreter can find the modules inside it.

It can contain initialization code for the Package, such as importing submodules, defining variables, or executing other code.

Syntax of Importing and Using __Init__.py File

To import a package or module, use the 'import' keyword followed by the package or module name. For instance, importing module1 from the 'package' package is done with:

```
import mypackage.module1
```

Alternatively, use 'from' followed by the package/module name, and 'import' for specific functions, classes, or variables. Example:

```
from mypackage.module1 import func1
```

__init__.py is a special Python file that turns a directory into a package.
- It can be empty or contain initialization code.
- It controls package imports and exposure of modules.
- Before Python 3.3 it was mandatory, now optional due to namespace packages, but still widely used.

what are python built-in modules(os,sys,datetime)?

Module	Purpose	Common Use Cases
os	OS-level operations	File handling, environment variables, directory management
sys	Interpreter control	Command-line args, exiting, modifying module search path

datetime	Date/time handling	Scheduling, logging, formatting, time arithmetic

1. os Module

👉 Used for interacting with the Operating System.

- File and directory operations.
- Environment variables.
- Process management.

eg.

```
import os
print(os.name)              # 'posix' (Linux/Mac), 'nt' (Windows)
print(os.getcwd())          # Get current working directory
os.mkdir("test_dir")        # Create new directory
```

2. sys Module

👉 Used to interact with the Python interpreter itself.

- Access command-line arguments.
- Control Python runtime environment.
- Manage Python path.

eg.

```
import sys
print(sys.version)      # Python version
print(sys.argv)         # Command-line arguments (list)
sys.path.append("/mydir") # Add a new path for module search
sys.exit()              # Exit the program
```

- Very useful in CLI tools and debugging.
- sys.argv is frequently asked (how to pass arguments to Python script).

3. datetime Module

👉 Used for working with dates and times.

- Getting current time.
- Formatting dates.
- Doing date arithmetic.

Eg

```
import datetime
now = datetime.datetime.now()
print(now)                  # 2025-10-01 12:30:45.123456
print(now.strftime("%Y-%m-%d"))  # '2025-10-01'
```

Used in logging, scheduling, time-stamping, and data analytics.

how do you open file in python?

Opening a File in Python. We use the built-in open() function:

open(file, mode)

- file → path of the file (string)
- mode → tells Python how to open the file

Example 1: Reading a File

```
f = open("sample.txt", "r")
content = f.read()
print(content)
f.close()
```

Interview Points

1. Why use with open()?
    - It's safer → file closes automatically, even if error occurs.
2. What happens if file not found?
    - open("file.txt", "r") → raises FileNotFoundError.

3. Difference between "w" and "a"?
    - "w" overwrites, "a" appends.

difference between with open () and normal open().

- with open() is preferred because it uses a context manager, which guarantees file closure, prevents resource leaks, and makes code cleaner.
- Normal open() works but requires explicit close(), which is error-prone if exceptions occur.

Example 1: Normal open()

```
f = open("sample.txt", "r")
data = f.read()
print(data)
f.close()  # MUST close manually
```

If you forget f.close(), the file stays open → can lock the file or waste system resources.

Example 2: with open() (Better Way)

```
with open("sample.txt", "r") as f:
    data = f.read()
    print(data)  # file auto-closes here
```

No need to call f.close(). Python automatically closes the file when the block ends.

Aspect	open() (normal)	with open() (context manager)
Closing the File	You must close manually using f.close(). If you forget, file may stay open.	Automatically closes file when block ends (even if an error occurs).
Error Handling	If exception occurs before f.close(), file may remain open → memory leak.	Ensures safe handling. File is closed automatically using context manager protocol.
Readability	Slightly more boilerplate.	Cleaner, more Pythonic.
Best Practice	Not recommended for production unless file is very simple.	Recommended for all real-world usage.

how to read /write JSON files in python?

Reading and writing JSON files in Python is straightforward using the built-in json module. Here's a clear explanation with examples.

1. **Import the json module**

```
import js
```

2. Writing JSON to a file

Suppose you have a Python dictionary that you want to save as a JSON file:

```
data = {
    "name": "Swapnil",
    "age": 28,
    "skills": ["Python", "React", "Java"]
}
```

 # Write to a file

```
with open("data.json", "w") as json_file:
    json.dump(data, json_file, indent=4)  # indent=4 makes it pretty-printed
```

✅ This will create a file data.json with content:

```
{
    "name": "Swapnil",
    "age": 28,
    "skills": [
        "Python",
```

```
        "React",
        "Java"
    ]
}
```

3. Reading JSON from a file

```
with open("data.json", "r") as json_file:
    data = json.load(json_file)

print(data)
print(data["name"])   # Swapnil
print(data["skills"]) # ['Python', 'React', 'Java']
```

difference between error and exceptions in python ?

1. Errors

- Definition: Errors are problems in the code that usually cannot be handled by the program and are detected at compile-time or runtime.
- Cause: Syntax mistakes, incorrect usage of Python features, or serious problems like memory overflow.
- Examples:
    - SyntaxError – Wrong Python syntax.
    - IndentationError – Incorrect indentation.
    - MemoryError – System ran out of memory.

Example:

```
# SyntaxError
print("Hello World"
```

Output: SyntaxError: unexpected EOF while parsing

---

2. Exceptions

- Definition: Exceptions are runtime events that can potentially be handled by the program using try-except blocks.
- Cause: Logical errors, invalid operations, or unexpected events during program execution.
- Examples:
    - ZeroDivisionError – Division by zero.
    - FileNotFoundError – Trying to open a non-existent file.
    - ValueError – Invalid value type.

Example:

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

Output: Cannot divide by zero!

Feature	Error	Exception
Occurrence	Usually detected at compile-time	Occurs at runtime
Handling	Cannot be handled	Can be handled using try-except
Example	SyntaxError, IndentationError	ZeroDivisionError, ValueError
Cause	Faulty code or system issue	Invalid operations or unexpected events

- Errors → Serious problems, mostly unhandled, indicate bugs.
- Exceptions → Runtime issues that can be caught and handled gracefully.

Assert Keyword in Python

In simpler terms, we can say that assertion is the boolean expression that checks if the statement is True or False. If the statement is true then it does nothing and continues the execution, but if the statement is False then it stops the execution of the program and throws an error.
In Python, assert is a statement used for debugging and for making sure that certain conditions hold true during the execution of your code.

Example 1: Simple Assertion

```
x = 5
assert x > 0
print("x is positive")
```

Output:
x is positive
Example 2: Assertion Failure

```
x = -3
assert x > 0, "x must be positive"
```

Output:
AssertionError: x must be positive

- Assertions can be disabled when running Python with the -O (optimize) flag.
- Validate conditions → Ensure variables meet expected conditions.
- Should not be used for regular runtime error handling (use if or exceptions for that).
- Prevent invalid operations → Catch programming errors early.

Try/Except/Finally block usage in python?

Using try/except/finally is crucial in Python to handle runtime errors gracefully and release resources properly (like files, network connections).
In Python, try / except / finally blocks are used for exception handling, allowing your program to handle errors gracefully instead of crashing. Here's a detailed explanation:
Syntax

```
try:
   # Code that may raise an exception
except SomeException as e:
   # Code to handle the exception
finally:
   # Code that will always run, whether exception occurred or not
```

- try → Block of code to test for exceptions
- except → Block to handle specific exception(s)
- finally → Block that runs always, useful for cleanup
-

Basic Example

```
try:
   x = int(input("Enter a number: "))
   result = 10 / x
except ZeroDivisionError:
   print("Error: Cannot divide by zero!")
except ValueError:
   print("Error: Invalid input! Enter a number.")
finally:
   print("This block runs no matter what.")
```

Sample Output 1 (valid input):

```
Enter a number: 2
This block runs no matter what.
```

Sample Output 2 (division by zero):
Enter a number: 0
Error: Cannot divide by zero!
This block runs no matter what.

Key Points

1. Multiple except blocks → Catch different types of exceptions.
2. except Exception as e → Catch any exception and get its message.
3. finally → Always executes, even if return or break occurs in try.

what is decorators ? Give an example

A decorator in Python is a function that takes another function as input and extends or modifies its behavior without changing its source code.
They are commonly used for:

- Logging
- Authorization
- Caching
- Decorators are higher-order functions (functions that accept/return functions).
- You can stack multiple decorators on a function.
- Built-in decorators include @staticmethod, @classmethod, and @property.

A decorator is just a function that adds extra behavior to another function — without changing the original function's code.

Simple Example (No Arguments)

```
def my_decorator(func):
    def wrapper():
        print("🍕 Before the function runs")
        func()
        print("🍔 After the function runs")
    return wrapper

@my_decorator   # This means: say_hello = my_decorator(say_hello)
def say_hello():
    print("Hello!")
say_hello()
```

Step by step what happens:
You write @my_decorator above say_hello.
Python does this internally:
say_hello = my_decorator(say_hello)

1.
2. So now, say_hello is replaced with the wrapper function inside my_decorator.
3. When you call say_hello():
    - It first prints "🍕 Before the function runs"
    - Then it runs the original say_hello function → prints "Hello!"
    - Finally, it prints "🍔 After the function runs"

Output:
🍕 Before the function runs
Hello!
🍔 After the function runs

Explain python garbage collection?

Garbage Collection (GC) in Python is the process of automatically freeing memory that is no longer being used by the program.
This prevents memory leaks and keeps your program efficient.
Python mainly uses Reference Counting and a Cyclic Garbage Collector to manage memory.

- Garbage Collection (GC) in Python is the process of automatically freeing memory that is no longer being used by the program.
- This prevents memory leaks and keeps your program efficient.
- Python mainly uses Reference Counting and a Cyclic Garbage Collector to manage memory.

Reference counting

Python uses reference counting to manage memory. Each object keeps track of how many references point to it. When the reference count drops to zero i.e., no references remain, Python automatically deallocates the object.

```
import sys

x = [1, 2, 3]
print(sys.getrefcount(x))

y = x
print(sys.getrefcount(x))

y = None
print(sys.getrefcount(x))
```

Output
2
3
2
Explanation:

x is referenced twice initially (once by x, once by getrefcount()).

Assigning y = x increases the count.

Setting y = None removes one reference.

Problem with Reference Counting

Reference counting fails in the presence of cyclic references i.e., objects that reference each other in a cycle. Even if nothing else points to them, their reference count never reaches zero. Example:

```
import sys
x = [1, 2, 3]
y = [4, 5, 6]

x.append(y)
y.append(x)
print(sys.getrefcount(x))
print(sys.getrefcount(y))
```

output
3
3
Explanation:

x contains y and y contains x.

Even after deleting x and y, Python won't be able to free the memory just using reference counting, because each still references the other.

Garbage collection for Cyclic References

Garbage collection is a memory management technique used in programming languages to automatically reclaim memory that is no longer accessible or in use by the application. To handle such circular references, Python uses a Garbage Collector (GC) from the built-in gc module. This collector is able to detect and clean up objects involved in reference cycles.

To handle cycles, Python also has a cyclic garbage collector (in the gc module).

- It detects groups of objects that reference each other but are no longer accessible from your program.
- It then frees that memory.

👉 Example:

```
import gc
gc.collect()  # Forces garbage collection cycle
```