Table of Contents

No.	Questions

	What are the common built-in data types in Python?
	How We can get the data type of any object?
	Primitive/Non-Primitive Data Structures?

	What is If Else?
	Python While Loops
	Switch/Match Statements?
	Switch/Match Statements Using class?
	Switch/Match Statements Using function?

	Type Casting/Type Conversion?

	Copy Object
	Using Equal(=) Oprater
	Using Deep Copy
	Using shallow Copy
	Difference between Deep Copy and Shallow Copy in Python?

	What is Decorators?
	What are pickling and unpickling in Python?
	What is Python JSON?
	What is Monkey Patching?
	What is Lambda/Anonymous Function?
	What is Magic Method Or Dunder Methods?
	floor() and ceil() Functions?
	What is Generator Functions?
	What is *args and **kwargs in Python?
	What do *(single asterisk) and **(double asterisk)
	Positional (non-keyword) arguments

No.	Questions
	Why use *args and **kwargs in Python?
	Difference between *args and **kwargs in Python?
	Keyword arguments
	Arbitrary Arguments
	Exception Handling
	DataTime Function
	What is File Handling in Python?
	What is an Iterable?
	What is Iterators?
	Difference between Iterators and iterable?
	Difference between generator and iterators in python?
	Logging
	Math Module
	Multithreading
	Python Closures

Ques. What is Python?

- Python is a high-level, interpreted, general-purpose programming language.
- Python is an **interpreter** language. It means it executes the code line by line, which helps in debugging and testing.
- It was created by **Guido van Rossum**, and released in **1991**
- It is used for:
    - web development (server-side)
    - software development
    - mathematics
    - system scripting

Ques. Features of Python?

1. **Easy:-** Python is very easy to learn and understand; using this Python tutorial, any beginner can understand the basics of Python.
2. **Interpreted:-** It is interpreted(executed) line by line. This makes it easy to test and debug.
3. **Object-Oriented:-** The Python programming language supports classes and objects. We discussed these above.
4. **Free and Open Source:-** The language and its source code are available to the public for free; there is no need to buy a costly license.
5. **Portable:-** Since it is open-source, you can run Python on Windows, Mac, Linux or any other platform. Your programs will work without needing to the changed for every machine.
6. **GUI Programming:-** You can use it to develop a GUI (Graphical User Interface). One way to do this is through Tkinter.
7. **Large Library:-** Python provides you with a large standard library. You can use it to implement a variety of functions without needing to reinvent the wheel every time. Just pick the code you need and continue. This lets you focus on other important tasks.

Ques. Python Frameworks?

- **Web Development:** Django, Pyramid, Bottle, Tornado, Flask, web2py
- **GUI Development:** tkInter, PyGObject, PyQt, PySide, Kivy, wxPython
- **Scientific and Numeric:** SciPy, Pandas, IPython
- **Software Development:** Buildbot, Trac, Roundup
- **System Administration:** Ansible, Salt, OpenStack, xonsh

Ques. What is PEP 8?

- PEP stands for **Python Enhancement Proposal**.
- It is a set of rules that specify how to format Python code for maximum readability.
- PEP8 is a document that provides various guidline to write the readable in python.
- PEP8 describe how the developers can write the beautiful code.

Ques. How to get Id of any object?

- **id()** function takes a single parameter object.

```
a = 5
print(id(a))
```

Ques. a=1, b=1 does both have same Id or not?

- Two variables in Python have same id, but not in lists.

```
a = 10
b = 10
print(id(a))
print(id(b))

Output:-
9788992
9788992
```

- In List

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a is b)

Output:- False
```

- In tuples

```
a = (1, 2, 3)
b = (1, 2, 3)
print(a is b)

Output:- True
```

Ques. Is indentation required in python?

Indentation is necessary for Python. It specifies a block of code. All code within loops, classes, functions, etc is specified within an indented block. It is usually done using four space characters. If your code is not indented necessarily, it will not execute accurately and will throw errors as well.

Ques. File Extensions in Python?

- **.py–** The normal extension for a Python source file
- **.pyc-** The compiled bytecode
- **.pyd-** A Windows DLL file
- **.pyo-** A file created with optimizations
- **.pyw-** A Python script for Windows
- **.pyz-** A Python script archive

Ques. What is the difference between .py and .pyc files?

- **.py** files contain the source code of a program. Whereas, **.pyc** file contains the bytecode of your program.
- We get bytecode after compilation of **.py** file (source code). **.pyc** files are not created for all the files that you run. It is only created for the files that you import.

Ques. What is an Interpreted language?

- An Interpreted language **executes** its statements **line by line**. Languages such as Python, Javascript, R, PHP, and Ruby are prime examples of Interpreted languages.

Ques. What is a dynamically typed language?

- Type-checking can be done at two stages:-

1. **Static-** Data Types are checked before execution.
2. **Dynamic-** Data Types are checked during execution. Python is an interpreted language, executes each statement line by line and thus type-checking is done on the fly, during execution. Hence, Python is a Dynamically Typed Language.

Ques. Python Comments?

- Comments are hints that we add to our code to make it easier to understand.
- Comments are short descriptions along with the code to increase its readability.
- **single Line Comments:-** Comments starts with a #, and Python will ignore them:

```
#This is a comment
print("Hello, World!")

print("Hello, World!") #This is a comment
```

Multi Line Comments(OR)Docstring

- To add a multiline comment you could insert a # for each line.

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

- you can add a multiline string (triple quotes) in your code, and place your comment inside it.

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Ques. What is Docstrings?

- In Python, docstrings are a way of documenting modules, classes, functions, and methods. They are written within triple quotes (""" or ''') and can span multiple lines.
- Python docstrings are strings used right after the definition of a function, method, class, or module. They are used to document our code.
- There are two main types of docstrings:
    - **Single-line docstrings:** Used for simple explanations that fit on one line.
    - **Multi-line docstrings:** Used for more detailed explanations, including parameter descriptions, return values, and exceptions.
- **Example and Accessing Docstrings:-**

```
class Calculator:
    """
    A simple calculator class to perform basic arithmetic operations.
    """
    def add(self, a, b):
        """
        add(a, b): Return the sum of two numbers.
        """
        return a + b

    def multiply(self, a, b):
        """
        multiply(a, b): Return the product of two numbers.
        """
        return a * b

# Accessing the class docstring
print(Calculator.__doc__) # Output:- A simple calculator class to perform basic
arithmetic operations.

# Accessing the method docstring
print(Calculator.add.__doc__)     # Output:- add(a, b): Return the sum of two
numbers.
print(Calculator.multiply.__doc__)  # Output:-  multiply(a, b): Return the product
of two numbers.
```

```
def square(n):
    '''Take a number n and return the square of n.'''
    return n**2

print(square.__doc__)

Output:- Take a number n and return the square of n.
```

Ques. how to get docstring in python?

- Using the **doc** attribute:

```
def my_function():
    """This is a docstring for my_function."""
    pass

print(my_function.__doc__)

Output:- This is a docstring for my_function.
```

- Using inspect.getdoc():

```
import inspect

def my_function():
    """
        This is a docstring for my_function
        with indentation.
    """
    pass

print(inspect.getdoc(my_function))

Output:-
This is a docstring for my_function
with indentation.
```

Ques. Difference between Python comments and docstrings?

- **Comments:** Comments in Python start with a hash mark (#) and are intended to explain the code to developers. They are ignored by the Python interpreter.
- **Docstrings:** Docstrings provide a description of the function, method, class, or module. Unlike comments, they are not ignored by the interpreter and can be accessed at runtime using the .**doc** attribute.

Ques. What is python Variables?

- Variables are containers for storing data values.
- A variable name must **start** with a **letter** or the **underscore** character.
- A variable name **cannot** start with a **number**.
- Variable names are **case-sensitive** (age, Age and AGE are three different variables)

```
#Legal variable names:
myvar = "John"
MYVAR = "John"
my_var = "John"
myVar = "John"
myvar2 = "John"
_my_var = "John"

#Illegal variable names:
2myvar = "John"
my-var = "John"
my var = "John"
```

```
x = 5
y = "Mohit"
print(x)
print(y)

Output:-
5
Mohit
```

- **Single or Double Quotes:-** String variables can be declared either by using single or double quotes:

```
x = "John"
print(x)
#double quotes are the same as single quotes:
x = 'John'
print(x)

Output:-
John
John
```

- **Assign Multiple Values:** Python allows you to assign values to multiple variables in one line.

```
x, y, z = "Orange", "Banana", "Cherry"
```

```
print(x)
print(y)
print(z)

Output:-
Orange
Banana
Cherry
```

- **One Value to Multiple Variables:** And we can assign the same value to multiple variables in one line.

```
x = y = z = "Orange"

print(x)
print(y)
print(z)

Output:-
Orange
Orange
Orange
```

- **Variables Casting:** We want to specify the data type of a variable, this can be done with casting. and We can **get the data type** of a variable with the **type()** function.

```
x = str(3)
y = int(3)
z = float(3)

print(x)
print(y)
print(z)

Output:-
3
3
3.0
```

```
x = 5
y = "John"
print(type(x))
print(type(y))

Output:-
<class 'int'>
<class 'str'>
```

- **Unpack a Collection:** If we have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits

print(x)
print(y)
print(z)

Output:-

apple
banana
cherry
```

```
# double quotes are the same as single quotes:
x = 4 # x is of type int
x = "saxena" # x is now of type str
x = 'mohit' # x is now of type str
print(x)

Output:-Mohit
```

```
x, y, z = "Orange", "Banana", "Cherry"
   print(x)
   print(y)
   print(z)

output:- Orange<br>Banana<br>Cherry

# Assign Value to Multiple Variables
x = y = z = "Orange"
print(x)
print(y)
print(z)

Output:-
Orange
Orange
Orange
```

Output Variables(combine both text and a variable)

```
x = "awesome"
print("Python is " + x)
```

```
output:- Python is awesome

# Example 2
-----------
x = "Python is "
y = "awesome"
z =  x + y
print(z)

output:-Python is awesome
```

```
x = 5
y = 10
print(x + y)
```

output:- 15

Note:- If you try to combine a string and a number, Python will give you an error:

```
x = 5
y = "John"
print(x + y)
output:- TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Ques. Global Variables?

- Variables that are created outside of a function.
- Global variables can be used by everyone, both inside of functions and outside.

```
x = "awesome"
def myfunc():
  print("Python is " + x)
myfunc()

output:- Python is awesome
```

```
x = "awesome"
def myfunc():
  x = "fantastic"
  print("Python is " + x)
myfunc()
print("Python is " + x)

output:-
Python is fantastic
Python is awesome
```

Ques. What is the operator?

1. Arithmetic Operators
2. Comparison operators
3. Assignment operators
4. [Logical operators]
5. [Bitwise operators]
6. [Membership operators]
7. [Identity operators]

Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x-y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Operators	Descrption	Result
Addition(+)	Adds the values on either side of the operator.	3+4=7
Subtraction(-)	Subtracts the value on the right from the one on the left.	3+4=-1
Multiplication(*)	Multiplies the values on either side of the operator.	3*4=12
Division(/)	Divides the value on the left by the one on the right. Notice that division results in a floating-point value.	3/4=0.75
Exponentiation(**)	Raises the first number to the power of the second.	3**4=81
Floor Division(//)	Divides and returns the integer value of the quotient. It dumps the digits after the decimal.	10//3=3
Modulus(%)	Divides and returns the value of the remainder.	3%4=3

Comparison(Relational) Operator

Operators	Descrption	Result
Less than(<)	This operator checks if the value on the left of the operator is lesser than the one on the right.	3<4=True
Greater than(>)	It checks if the value on the left of the operator is greater than the one on the right.	3>4=False
Less than or equal to(<=)	It checks if the value on the left of the operator is lesser than or equal to the one on the right.	7<=7 = True
Greater than or equal to(>=)	It checks if the value on the left of the operator is greater than or equal to the one on the right.	0>=0 = True
Equal to(==)	This operator checks if the value on the left of the operator is equal to the one on the right.(1 is equal to the Boolean value True, but 2 isn't. Also, 0 is equal to False.)	3==3.0 = True 1==True = True 7==True = False 0==False = True 0.5==True = False

Assignment Operators

- Assignment operators

Operator	Example	Same As	Try it
=	x = 5	x = 5	x = 5 print(x) 5
+=	x += 3	x = x + 3	x = 5 x += 3 print(x) 8
-=	x -= 3	x = x - 3	x = 5 x -= 3 print(x) 2
*=	x *= 3	x = x * 3	x = 5 x *= 3 print(x) 15
/=	x /= 3	x = x/3	x = 5 x /= 3 print(x) 1.66

```
%=  x %= 3  x = x % 3
//= x //= 3 x = x // 3
**= x **= 3 x = x ** 3
&=  x &= 3  x = x & 3
|=  x |= 3  x = x | 3
^=  x ^= 3  x = x ^ 3
>>= x >>= 3 x = x >> 3
<<= x <<= 3 x = x << 3
```

Operators	Descrption	Result

Assign(=)	Assigns a value to the expression on the left. Notice that = = is used for comparing, but = is used for assigning.	>>> a=7
		>>>
		print(a)
		//output:-
		7

Add and Assign(+=)	Adds the values on either side and assigns it to the expression on the left. a+=10 is the same as a=a+10.	>>> a+=2

Operators	Descrption	Result
>>> print(a) //output:- 9		
Divide and Assign(/=)	Divides the value on the left by the one on the right. Then it assigns it to the expression on the left.	>>> a/=7 >>> print(a) //output:- 1.0
Multiply and Assign(*=)	Multiplies the values on either sides. Then it assigns it to the expression on the left.	>>> a*=8 >>> print(a) // 8.0
Modulus and Assign(%=)	Performs modulus on the values on either side. Then it assigns it to the expression on the left.	>>> a%=3 >>> print(a) //output:- 2.0
Exponent and Assign(**=)	Performs exponentiation on the values on either side. Then assigns it to the expression on the left.	>>> a**=5 >>> print(a) //output:- 32.0
Floor-Divide and Assign(//=)	Performs floor-division on the values on either side. Then assigns it to the expression on the left.	>>> a//=3 >>> print(a) //output:- 10.0

Bitwise Operators

Operators	Descrption	Result
Binary AND(&)	It performs bit by bit AND operation on the two values. Here, binary for 2 is 10, and that for 3 is 11. &-ing them results in 10, which is binary for 2.	>>> 2&3 //output:-2
Binary OR(	)	--
Binary XOR(^)	--	--
Binary One's Complement(~)	--	--
Binary Left-Shift(<<)	--	--
Binary Right-Shift(>>)	--	--

What is membership operators?

- A membership operator in Python checks if a value or variable is present in a sequence(string, list, tuples, sets, dictionary) or not.
- The output of a membership operator is a Boolean value, either True or False.
- There are two membership operators in Python.

1. In operator
2. not in' operator

in operator

- The 'in' operator is used to check if a value exists in a sequence or not.

```
#in
x = "Hello, World!"
print("ello" in x) #Returns true as it exists
print("hello" in x) #Returns false as 'h' is in lowercase
print("World" in x)#Returns true as it exists
```

```
x = ["apple", "banana"]
print("banana" in x)

Output:- True
```

```
list1=[0,2,4,6,8]
list2=[1,3,5,7,9]

check=0
for item in list1:
    if item in list2:
        #Overlapping true so check is assigned 1
        check=1

if check==1:
    print("overlapping")
else:
    print("not overlapping")

Output:- not overlapping
```

2. **'not in operator-** Returns True if the target value is not present in a given collection of values. Otherwise, it returns False.

```
#not in
x = "Hello, World!"
print("ello" not in x) #Returns false as it exists
print("hello" not in x) #Returns true as it does not exist
print("World" not in x) #Returns false as it exists
```

```
x = ["apple", "banana"]
print("pineapple" not in x)

Output:- True
```

```
a = 70
b = 20
list = [10, 30, 50, 70, 90 ];

if ( a not in list ):
 print("a is NOT in given list")
else:
 print("a is in given list")

if ( b not in list ):
 print("b is NOT present in given list")
else:
  print("b is in given list")

Output:-
a is in given list
b is NOT present in given list
```

What is Identity operators?

- The Python Identity Operators are used to compare the objects if both the objects are actually of the **same data type** and share the **same memory location**. There are different identity operators such as: There are two Identity operators in Python.

1. is operator
2. is not operator

is operator

- is operator returns True if both variables refer to the same object, otherwise, it returns False.

```
x = 'Educative'
if (type(x) is str):
    print("true")
else:
    print("false")
Output:- True
```

is not operator

- is not operator returns True if both variables do not refer to the same object, otherwise, it returns False.

```
x = 6.3
if (type(x) is not float):
    print("true")
else:
    print("false")

Output:- False
```

```
# is not Operator:
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is not z)   # Output:- False
print(x is not y)   # Output:- True
print(x != y)   # Output:- False
```

Difference between '==' and 'is' Operator?

- The equality operator(==) is used to compare the value of two variables, whereas the identities operator is used to compare the memory location of two variables.

Ques. What is Scope in Python?

- scope resolution in python follows the LEGB rules.

1. Local(L): Defined inside function/class
2. Enclosed(E): Defined inside enclosing functions(Nested function concept)
3. Global(G): Defined at the uppermost level
4. Built-in(B): Reserved names in Python builtin modules

- Every object in Python functions within a scope. A scope is a block of code where an object in Python remains relevant. Namespaces uniquely identify all the objects inside a program.

1. **Local Scope/Local Variables:-** The Variables which are defined in the function are a local scope of the variable. These variables are defined in the function body.

```
x = "awesome"

def myfunc():
    x = "fantastic"
    print("Python is " + x)
myfunc()
print("Python is " + x)

Output:-
Python is fantastic
Python is awesome
```

2. **Global Scope/Global Variables:-** The Variable which can be read from anywhere in the program is known as a global scope. These variables can be accessed inside and outside the function.

```
x = 300
def myfunc():
    print(x)
myfunc()

print(x)
Output:- 300
300
```

3. **NonLocal or Enclosing Scope:-** Nonlocal Variable is the variable that is defined in the nested function. It means the variable can be neither in the local scope nor in the global scope.

```
def func_outer():
    x = "local"
    def func_inner():
        nonlocal x
        x = "nonlocal"
```

```
        print("inner:", x)
    func_inner()
    print("outer:", x)
func_outer()

Output:-
inner: nonlocal
outer: nonlocal
```

4. **Built-in Scope:-** If a Variable is not defined in local, Enclosed or global scope, then python looks for it in the built-in scope. In the Following Example, 1 from math module pi is imported, and the value of pi is not defined in global, local and enclosed. Python then looks for the pi value in the built-in scope and prints the value. Hence the name which is already present in the built-in scope should not be used as an identifier.

```
# Built-in Scope
from math import pi
# pi = 'Not defined in global pi'
def func_outer():
    # pi = 'Not defined in outer pi'
    def inner():
        # pi = 'not defined in inner pi'
        print(pi)
    inner()
func_outer()

Output:- 3.141592
```

Ques. global Keyword?

- To create a global keyword inside a function should be treated as a global variable, you can use the global keyword.

```
def myfunc():
  global x
  x = "fantastic"

myfunc()
print("Python is " + x)
Output:- Python is fantastic
```

- Also, use the global keyword if you want to change a global variable inside a function.

```
x = 10  # Global variable
def modify_global():
    global x
    x = 20  # Changing the value of the global variable

modify_global()
print(x)  # This will print 20
```

Ques. What are the common built-in data types in Python?

- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

1. **Text Type:-** str
2. **Numeric Types:-** int, float, complex
3. **Sequence Types:** list, tuple, range
4. **Mapping Type:** dict
5. **Set Types:** set, frozenset
6. **Boolean Type:** bool
7. **Binary Types:** bytes, bytearray, memoryview
8. **None Type:** NoneType

- **Numeric:-**
    - Integers :- int stores integers eg a=100, b=25, c=526, etc.
    - Float :- float stores floating-point numbers eg a=25.6, b=45.90, c=1.290, etc.
    - Complex Numbers:- complex stores numbers eg a=3 + 4j, b=2 + 3j, c=complex(4,6), etc.
    - long:- long stores higher range of integers eg a=908090999L, b=-0x1990999L, etc.
- **Sequence Type:-**
    - String
    - List
    - Tuple
    - range
- **Boolean:-** There can be only two types of value in the Boolean data type of Python, and that is True or False.
- **Set Type:-**
    - set:-
    - frozenset:-
- **Dictionary**
- **long:-** long stores higher range of integers eg a=908090999L, b=-0x1990999L, etc.
- **Mapping Types:-**
    - dict:- Stores comma-separated list of key: value pairs.
- **Binary Types:-**
    - bytes
    - bytearray
    - memoryview

Ques. How We can get the data type of any object?

- By using the **type()** function.

```
x = 5
print(type(x))

output:- <class 'int'>
```

Ques. Primitive/Non-Primitive Data Structures

- Primitive Data Structures
  - Integers
  - Float
  - Strings
  - Boolean
- Non-Primitive Data Structures
  - Arrays
  - Lists
  - Tuples
  - Dictionary
  - Sets

Ques. What is If Else?

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
output:- a is greater than b
```

- **Short Hand If**

```
a = 200
b = 33

if a > b: print("a is greater than b")  # output:- "a is greater than b"
```

- **Short Hand If ... Else**

```
a = 2
b = 330

print("A") if a > b else print("B") # output:- B
```

- One line if else statement, **with 3 conditions:**

```
a = 330
b = 330

print("A") if a > b else print("=") if a == b else print("B")
Output:- =
```

- The **and** keyword is a logical operator, and is used to combine conditional statements.

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
output:- Both conditions are True
```

- The **Or** keyword is a logical operator, and is used to combine conditional statements.

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
output:- At least one of the conditions is True
```

- **Nested If** You can have if statements inside if statements, this is called nested if statements.

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
Output:-
Above ten,
and also above 20!
```

- **The pass Statement** if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200

if b > a:
  pass
Output:-
```

Ques. Python While Loops?

- With the while loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i += 1

output:-
1
2
3
4
5
```

- **The break Statement:-** With the break statement we can stop the loop even if the while condition is true:

```
i = 1
while i < 6:
  print(i)
  if (i == 3):
    break
  i += 1

Output:-
1
2
3
```

- **The continue Statement:-** With the continue statement we can stop the current iteration, and continue with the next.

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)

Output:-
1
2
4
5
6
```

Ques. Switch Statements?

- The match statement is used to perform different actions based on different conditions.
- Instead of writing many **if..else statements**, you can use the match statement.

```
lang = input("What's the programming language you want to learn? ")

match lang:
    case "JavaScript":
        print("You can become a web developer.")

    case "Python":
        print("You can become a Data Scientist")

    case "PHP":
        print("You can become a backend developer")

    case "Solidity":
        print("You can become a Blockchain developer")

Output:-
What's the programming language you want to learn?  PHP
You can become a backend developer

What's the programming language you want to learn? node
The language doesn't matter, what matters is solving problems.
```

Default Value

- Use the **underscore character** _ as the last case value if you want a code block to execute when there are not other matches:

```
day = 4
match day:
  case 6:
    print("Today is Saturday")
  case 7:
    print("Today is Sunday")
  case _:
    print("Looking forward to the Weekend")

Output:-
Looking forward to the Weekend
```

Combine Values

- Use the pipe character | as an or operator in the case evaluation to check for more than one value match in one case:

```
day = 4
match day:
  case 1 | 2 | 3 | 4 | 5:
    print("Today is a weekday")
  case 6 | 7:
    print("I love weekends!")

Output:-
Today is a weekday
```

If Statements as Guards

- You can add if statements in the case evaluation as an extra condition-check:

```
month = 5
day = 4
match day:
  case 1 | 2 | 3 | 4 | 5 if month == 4:
    print("A weekday in April")
  case 1 | 2 | 3 | 4 | 5 if month == 5:
    print("A weekday in May")
  case _:
    print("No match")

Output:-
A weekday in May
```

```
subject = input("Enter a subject: ")
score = int(input("Enter a score: "))

match subject:
    # if score is 80 or higher in Physics or Chemistry
    case 'Physics' | 'Chemistry' if score >= 80:
        print("Excellent in Science!")

    # if score is 80 or higher in English or Grammar
    case 'English' | 'Grammar' if score >= 80:
        print("Excellent in English!")

    # if score is 80 or higher in Maths
    case 'Maths' if score >= 80:
        print("Excellent in Maths!")

    case _:
        print(f"Needs improvement in {subject}!")


Output:-
```

```
Enter a subject: Physics
Enter a score: 2
Needs improvement in Physics!
```

Switch Statements Using class

```
class Python_Switch:
    def day(self, month):

        default = "Incorrect day"
        return getattr(self, 'case_' + str(month), lambda: default)()

    def case_1(self):
        return "Jan"

    def case_2(self):
        return "Feb"

    def case_3(self):
        return "Mar"

my_switch = Python_Switch()

print(my_switch.day(1))
print(my_switch.day(3))

Output:-
Jan
Mar
```

Switch Statements Using function

```
def number_to_string(argument):
    match argument:
        case 0:
            return "zero"
        case 1:
            return "one"
        case 2:
            return "two"
        case default:
            return "something"


head = number_to_string(2)
print(head)     # Output:- two
```

Ques. Type Casting/Type Conversion?

- The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion.
- When we convert the value of one data type to the value of another data type, we call this type conversion.
- Integers

```
x = int(1)
y = int(2.8)
z = int("3")
print(x)    # Output:- 1
print(y)    # Output:- 2
print(z)    # Output:- 3
```

- Floats

```
# Floats
x = float(1)
y = float(2.8)
z = float("3")
print(x)    # Output:- 1.0
print(y)    # Output:- 2.8
print(z)    # Output:- 3.0
```

- Strings

```
x = str("s1")
y = str(2)
z = str(3.0)
print(x)    # Output:- s1
print(y)    # Output:- 2
print(z)    # Output:- 3.0
```

Python has **two** types of type conversion.

1. **Implicit Type Conversion: -** In Implicit type conversion, Python **automatically converts** one data type to another data type. This process doesn't need any user involvement.

```
num_int = 123
num_flo = 1.23

num_new = num_int + num_flo

print(type(num_int))    # Output:- <class 'int'>
print(type(num_flo))    # Output:- <class 'float'>
print(num_new)          # Output:- 124.23
print(type(num_new))    # Output:- <class 'float'>
```

2. **Explicit Type Conversion: -** In Explicit Type Conversion, **users convert** the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))
num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))
num_sum = num_int + num_str
print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))

Output:-
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>
Data type of num_str after Type Casting: <class 'int'>
Sum of num_int and num_str: 579
Data type of the sum: <class 'int'>
```

Note:-

- If you try to **combine** a **string** and a **number**, Python will give you an **error**:

```
x = 5
y = "John"
print(x + y)
output:- TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Copy Of Object

Copy Object Using Equal(=) Oprater

- In Python, we use **=** operator to create a copy of an object. It only creates a new variable that shares the reference of the original object.
- When we make any **changes** to a copy **of** an **object**, those changes **do reflect** in the original object **because** it creates a new object that **stores** the **references** of the **original** elements.

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list

new_list[2][2] = 9

print('Old List:', old_list) # output:- Old List: [[1, 2, 3], [4, 5, 6], [7, 8,
9]]
print('New List:', new_list) # Output:- New List: [[1, 2, 3], [4, 5, 6], [7, 8,
9]]
```

Copy object Using Deep Copy

- In deep copy, When we make any changes to a copy of an object, those changes **do not reflect** in the **original object**.
- Deep copying in Python ensures that the copy is a completely independent object, with **no** shared **references** to the **original object** or its nested components. This is why changes made to a deep copy do not reflect in the original object.

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

new_list[1][0] = 'BB'

print("Old list:", old_list)
print("New list:", new_list)

Output:-
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
```

Copy object Using Shallow copy

- A shallow copy creates a new object which **stores** the **reference** of the **original elements**.
- When we make any changes to a copy of an object, those **changes** do reflect in the **original object** because it creates a new object that **stores** the **references** of the **original elements.**

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)

new_list[1][1] = 'AA'

print("Old list:", old_list)
print("New list:", new_list)

Output:-
Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

Difference between Deep Copy and Shallow Copy in Python?

Shallow Copy	Deep Copy
Creates a new object with a new reference, but the content is the same reference as the original object.	Creates a new object with a new reference, and the content is a new copy of the original object.
Shares reference to original object, changes to copy affects the original.	Independent from the original object, changes to copy do not affect the original.
Only copies the top-level elements of an object.	Copies all levels of nested objects.
Memory-efficient, as it doesn't create new objects for nested references.	Memory-intensive, as it creates new objects for nested references.
Fast operation, as it doesn't create new objects for nested references.	Slow operation, as it creates new objects for nested references.

Ques. What is Decorators?

- A decorator is a **design pattern** in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are usually called before the definition of a function you want to decorate.
- Decorators are used to add some design patterns to a function without changing its structure.
- A decorator function is a function that accepts a function as parameter and return a function(decorator ek function hai jo as a argument leta bhi function hai and return bhi function karta hai).
- Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.

```
# A simple decorator function
def decorator(func):

    def wrapper():
        print("Before calling the function.")
        func()
        print("After calling the function.")
    return wrapper

# Applying the decorator to a function
@decorator

def greet():
    print("Hello, Mohit!")

greet()

Output:-
Before calling the function.
Hello, Mohit!
After calling the function.
```

Explanation:

- decorator takes the greet function as an argument.
- It returns a new function (wrapper) that first prints a message, calls greet() and then prints another message.
- The @decorator syntax is a shorthand for greet = decorator(greet).

Syntax of Decorator Parameters

```
def decorator_name(func):
    def wrapper(*args, **kwargs):
        # Add functionality before the original function call
        result = func(*args, **kwargs)
        # Add functionality after the original function call
        return result
```

```
    return wrapper


@decorator_name
def function_to_decorate():
    # Original function code
    pass
```

Explanation of Parameters(Syntax of Decorator Parameters)

1. decorator_name(func):
    1. decorator_name: This is the name of the decorator function.
    2. func: This parameter represents the function being decorated. When you use a decorator, the decorated function is passed to this parameter.
2. wrapper(*args, **kwargs):
    1. wrapper: This is a nested function inside the decorator. It wraps the original function, adding additional functionality.
    2. *args: This collects any positional arguments passed to the decorated function into a tuple.
    3. **kwargs: This collects any keyword arguments passed to the decorated function into a dictionary.
    4. The wrapper function allows the decorator to handle functions with any number and types of arguments.
3. @decorator_name:
    1. This syntax applies the decorator to the function_to_decorate function. It is equivalent to writing function_to_decorate = decorator_name(function_to_decorate).

- 2nd Type

```
def decor(fun1):
    def inner():
        print('befor inhance')
        fun1()
        print('after inhance')
    return inner

@decor
def num():
    print('hello mohit')

num()

Output:-
befor inhance
hello mohit
after inhance
```

```
def num_decor(num):
    def inner():
        a = num()
        add = a + 5
        return add
    return inner

@num_decor
def num():
    return 10

print(num())

Output:- 15

# Example:-
def num_decor(num):
    def inner():
        a = num()
        add = a + 5
        return add
    return inner

def num():
    return 10

result = num_decor(num)
print(result())

Output:- 15
```

- Upper case decorater

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper

def say_hi():
    return 'hello there'

# call decorater
decorate = uppercase_decorator(say_hi)
print(decorate())

# -------2nd Opton call decorater----------
```

```
@uppercase_decorator
def say_hi():
    return 'hello there'

print(say_hi())
--------------------------

Output:-'HELLO THERE'
```

Higher-Order Functions

- In Python, higher-order functions are functions that take one or more functions as arguments, return a function as a result or do both. Essentially, a higher-order function is a function that operates on other functions.
- **Key Properties of Higher-Order Functions:**
    - Taking functions as arguments: A higher-order function can accept other functions as parameters.
    - Returning functions: A higher-order function can return a new function that can be called later.

```
# A higher-order function that takes another function as an argument
def fun(f, x):
    return f(x)

# A simple function to pass
def square(x):
    return x * x

# Using apply_function to apply the square function
res = fun(square, 5)
print(res)

# Output:
25
```

Ques. What are pickling and unpickling in Python?

- pickling and unpickling should be done using binary files since they support byte steam.
- Python pickle module is used for serializing and de-serializing python object structures. The process to converts any kind of python objects (list, dict, etc.) into byte streams (0s and 1s) is called pickling or serialization or flattening or marshalling. We can converts the byte stream (generated through pickling) back into python objects by a process called as unpickling. **Pickling:**
- pickling is a process of converting a class **object** into a **byte** stream so that it can be stored into a file. this is also called as object serialization.
- Pickling is the name of the serialization process in Python. Any object in Python can be serialized into a byte stream and dumped as a file in the memory.
- The function used for the above process is **pickle.dump()**.

unpickling:

- unpickling is a process whereby **byte** stream is converted back into a class **object**. it is inverse operation of pickling. this is also called as de-serialization.
- The function used for the above process is **pickle.load()**.

```
import pickle
class Student:
    def __init__(self, name, roll, address):
        self.name = name
        self.roll = roll
        self.address = address

    def display(self):
        print(f"name {self.name} roll is {self.roll} address {self.address}")

with open('student.dat', mode='wb') as f:
    stu1 = Student('mohit', 001, 'mainpuri')
    stu2 = Student('rohit', 001, 'agra')
    pickle.dump(stu1, f)
    pickle.dump(stu2, f)
    print('pickling Done')

with open('student.dat', mode='rb') as f:
    obj1 = pickle.load(f)
    obj2 = pickle.load(f)
    print('unpikling Done!!!')
    obj1.display()
    obj2.display()
```

Ques. What is Python JSON?

- JSON **JavaScript Object Notation** is a format for structuring data. It is mainly used for storing and transferring data between the browser and the server.
- Python has a built-in package called json, which can be used to work with JSON data.
- **Convert JSON to Python:-** If you have a JSON string, you can parse it by using the **json.loads()** method.

```
import json
# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'
y = json.loads(x)
# the result is a Python dictionary:
print(y["age"])

Output:- 30
```

- **Convert Python to JSON:-** If you have a Python object, you can convert it into a JSON string by using the **json.dumps()** method.

```
 import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)

Output:- {"name": "John", "age": 30, "city": "New York"}
```

- Format the Result:- Use the **indent** parameter to define the numbers of indents:

```
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
```

```
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

# use four indents to make it easier to read the result:
print(json.dumps(x, indent=4))

Output:-
{
    "name": "John",
    "age": 30,
    "married": true,
    "divorced": false,
    "children": [
        "Ann",
        "Billy"
    ],
    "pets": null,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ]
}
```

- Use the **sort_keys** parameter to specify if the result should be sorted or not:

```
print(json.dumps(x, indent=4, sort_keys=True))

Output:-
{
    "age": 30,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ],
    "children": [
```

```
        "Ann",
        "Billy"
    ],
    "divorced": false,
    "married": true,
    "name": "John",
    "pets": null
}
```

Ques. What is Monkey Patching?

- The term monkey patching refers to dynamic(or run time) modifictaion of class or method.
- A class or method can be changed at the runtime.

```
class A:
   def hello(self):
      print ("The hello() function is being called")

def monkey_f():
    print ("monkey_f() is being called")

#normal class method call
obj = A()
obj.hello()

#calling class method after monkey patch
obj.hello = monkey_f
obj.hello()

Output:-
The hello() function is being called
monkey_f() is being called
```

```
# Original buggy function
def calculate_area(length, width):
  return length * width  # Missing multiplication by 2

# Monkey patch the bug
def fixed_calculate_area(length, width):
  return length * width * 2

calculate_area = fixed_calculate_area

print(calculate_area(5, 3))  # Output: 30 (correctly calculated)
```

```
# Original Class
class Power:
    def square(self, num):
        return f"Square of {num} is: {num**2}"

# Creating an object of Power
obj = Power()
print(obj.square(3))  # Expected output: Square of 3 is: 9

------------apply moncky paching
# Define the replacement function
def cube(self, num):
```

```
    return f"Cube of {num} is: {num**3}"

# Monkey Patching
Power.square = cube  # Assigning the new 'cube' method to replace 'square'

# Testing the patch
obj = Power()
print(obj.square(3))  # Expected output: Cube of 3 is: 27
```

Ques. What is Lambda/Anonymous Function?

- A lambda function is a small anonymous function.
- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the **def** keyword in Python, anonymous functions are defined using the **lambda** keyword.
- **Notice** that the anonymous function does not have a return keyword. This is because the anonymous function will automatically return the result of the expression in the function once it is executed.

```
def add(a,b):
  print(a+b)
add(5,10)

# Using Lambda function
x = lambda a: a + 10
print(x(5))

Output:- 15
```

- You can use lambda function in **filter()**

```
# filter() function is used to filter a given iterable (list like object) using
another function that defines the filtering logic.
# Syntex:- filter(object, iterable)
# The object here should be a lambda function which returns a boolean value.
mylist = [2,3,4,5,6,7,8,9,10]
list_new  = list(filter(lambda x : (x%2==0), mylist))
print(list_new)

Output:- [2, 4, 6, 8, 10]
```

- We can use lambda function in **map()**

```
# map() function applies a given function to all the itmes in a list and returns
the result. Similar to filter(), simply pass the lambda function and the list (or
any iterable, like tuple) as arguments.

mylist = [2,3,4,5,6,7,8,9,10]
list_new  = list(map(lambda x : x%2, mylist))
print(list_new)

Output:- [0, 1, 0, 1, 0, 1, 0, 1, 0]
```

- You can use lambda function in **reduce()** as well

```
# reduce() function performs a repetitive operation over the pairs of the elements
in the list. Pass the lambda function and the list as arguments to the reduce()
function. For using the reduce() function, you need to import reduce from
functools librray.

from functools import reduce
list1 = [1,2,3,4,5,6,7,8,9]
sum = reduce((lambda x,y: x+y), list1)
print(sum)

Output:- 45 //i.e 1+2, 1+2+3 , 1+2+3+4 and so on.
-------------------------------------------------------------------------------
# How to use lambda function to manipulate a Dataframe
# You can also manipulate the columns of the dataframe using the lambda function.
It’s a great candidate to use inside the apply method of a dataframe. I will be
trying to add a new row in the dataframe in this section as example.

import pandas as pd
df = pd.DataFrame([[1,2,3],[4,5,6]],columns = ['First','Second','Third'])
df['Forth']= df.apply(lambda row: row['First']*row['Second']* row['Third'],
axis=1)
df

Output:-
|   | First | Second | Third | Forth |
| 0 |  1    |    2   |   3   |  6    |
| 1 |  4    |  5     |   6   |  120  |
```

Ques. What is Magic Method Or Dunder Methods?

- Python Magic methods are the methods starting and ending with double underscores '__'. They are also called Dunder methods, Dunder here means "Double Under (Underscores)".
- The dir() function can be used to see the number of magic methods inherited by a class.
- print(dir(int))

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Special Method	Description
len()	Supports the len() function

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

    def __len__(self):            # Magic method
        return len(self.name)

obj = emp('mohit', 422573)
print(obj.__len__())    # Output:- 5
print(len(obj))         # Output:- 5
```

Ques. floor() and ceil() Functions?

- **floor() Function:-** The nearest round-down number of it.

```
import math as M
print ("math.floor(-23.11) : ", M.floor(-23.11))
print ("math.floor(300.16) : ", M.floor(300.16))
print ("math.floor(300.72) : ", M.floor(301.72))

Output:-
math.floor(-23.11) :  -24
math.floor(300.16) :  300
math.floor(300.72) :  301
```

- **ceil() Function:-** The nearest round-up number of it.

```
import math as M
print ("math.ceil(-23.11) : ", M.ceil(-23.11))
print ("math.ceil(300.16) : ", M.ceil(300.16))
print ("math.ceil(300.72) : ", M.ceil(301.72))

Output:-
math.ceil(-23.11) :  -23
math.ceil(300.16) :  301
math.ceil(300.72) :  302
```

Ques. What is Generator Functions?

- Generator are functions that returns a sequens of value. we use **yield** statement to return the from function.
- Yield statement returns the element from a generator function into a genrater object.(EX:- yield a)
- This function is used to retrieve element by element from a generator object.(Ex:- next(gen_obj))
- A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values.
- In a generator function, a **yield** statement is used rather than a return statement.
- The generator function cannot include the return keyword. If you include it, then it will terminate the function.
- The **difference** between **yield** and **return** is that yield **returns a value and pauses the execution** while maintaining the internal states, whereas the **return statement returns a value and terminates the execution** of the function.

```
def mygenerator():
    print('First item')
    yield 10

    print('Second item')
    yield 20

    print('Last item')
    yield 30

gen = mygenerator()
print(next(gen))
print(gen.__next__())  # 2 option to write the next function
print(next(gen))
print(next(gen))

Output:-
First item
10
Second item
20
Last item
30
Traceback (most recent call last):
File "<string>", line 22, in <module>
StopIteration

------------------------------------------------------------
# 2nd Option
gen = mygenerator()
while True:
    try:
        print ("Received on next(): ", next(gen))
    except StopIteration:
        break
Output:-
```

```
First item
Received on next():  10
Second item
Received on next():  20
Last item
Received on next():  30

# Example 2:-
def bhai(a,b):
    yield a+b
    yield a-b
result = bhai(3,2)
print(next(result))
print(next(result))

Output:-
5
1

# Example 3:-
def numberPrint():
    n = 1
    while n <= 10:
        sq = n*n
        yield sq
        n += 1
values = numberPrint()
for i in values:
    print(i)

Output:-
1
4
9
16
25
36
49
64
81
100
```

Ques. What do * (single asterisk) and ** (double asterisk)?

Ques. What do * args and ** kwargs?

*args (Non-Keyword Arguments)

- *args allows you to pass a variable number of non-keyword arguments to a function.
- args is treated as a tuple containing all the extra arguments that were passed.

```
# Example 1
def print_colors(*args):
    print(args)
print_colors('red','blue','green','yellow')

Output:- ('red', 'blue', 'green', 'yellow')

# Example 2
def myFun(*argv):
    for arg in argv:
        print(arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')

Output:-
Hello
Welcome
to
GeeksforGeeks

# Example 3
def fun(arg1, *argv):
    print("First argument :", arg1)
    for arg in argv:
        print("Argument *argv :", arg)


fun('Hello', 'Welcome', 'to', 'GeeksforGeeks')

Output:-
First argument : Hello
Argument *argv : Welcome
Argument *argv : to
Argument *argv : GeeksforGeeks
```

****kwargs(Keyword Arguments)**

- kwargs allows you to pass a variable number of keyword arguments (arguments that have a **key-value pair**) to a function.
- kwargs is treated as a **dictionary** containing all the extra keyword arguments that were passed.

```
# Example 1
def print_numbers(**kwargs):
  for key, value in kwargs.items():
      print (f"{key} is a {value}")
print_numbers(mohit="TL", two="two",three=3,four="four")

Output:-
mohit is a TL
two is a two
three is a 3
four is a four

# Example 2
def fun(arg1, **kwargs):
    for k, val in kwargs.items():
        print("%s == %s" % (k, val))


# Driver code
fun("Hi", s1='Geeks', s2='for', s3='Geeks')

Output:-
s1 == Geeks
s2 == for
s3 == Geeks
```

Using both *args and **kwargs

```
def fun(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

fun(1, 2, 3, a=4, b=5)
Output:-
Positional arguments: (1, 2, 3)
Keyword arguments: {'a': 4, 'b': 5}
```

Why use *args and **kwargs in Python?

- *args and **kwargs allow functions to accept a variable number of arguments:
  - *args (arguments) allows you to pass a variable number of positional arguments to a function.
  - **kwargs (keyword arguments) allows you to pass a variable number of keyword arguments (key-value pairs) to a function.

Difference between *args and **kwargs in Python?

- *args collects additional positional arguments as a tuple, while **kwargs collects additional keyword arguments as a dictionary.

```
def example_function(*args, **kwargs):
    print(args)    # tuple of positional arguments
    print(kwargs)  # dictionary of keyword arguments

example_function(1, 2, 3, name='Alice', age=30)
Output:
(1, 2, 3)
{'name': 'Alice', 'age': 30}
```

Positional (non-keyword) arguments:

- These are the most common type of arguments. Their position in the function call determines which parameter they correspond to.

```
def greet(name, greeting):
    print(f"{greeting}, {name}!")

greet("Alice", "Hello") # "Hello, Alice!"
```

Keyword arguments:

- These arguments are **passed with the parameter name** explicitly specified. This allows you to pass arguments in any order and makes the code more readable.

```
def greet(name, greeting):
    print(f"{greeting}, {name}!")

greet(greeting="Hi", name="Bob") # "Hi, Bob!"
```

Arbitrary Arguments:

- If you do not know how many arguments will be passed to your function, add a * before the parameter name in the function definition. This way the function will receive a tuple of arguments, and can access

the items accordingly.

```
def my_function(*kids):
    print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus") # output The youngest child is Linus
```

- If you do not know how many keyword arguments that will be passed to your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly.

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes") # output His last name is Refsnes
```

Exception Handling

Try, Except, else and Finally

- **try:-** try block lets you test a block of code for errors.
- **except:-** The except block lets you handle the error.
- **else:-** The else block lets you execute code when there is no error.
- **finally:-** Finally block always gets executed either exception is generated or not

```
# Example_1 for try except
try:
  print(x)
except:
  print("Something went wrong")

Output:-
Something went wrong

# Example_2 for try except else
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("execute if no exception")

Output:-
Hello
execute if no exception


# Example_3 for try except else finally
try:
  print(x)
except:
  print("Something went wrong")
else:
  print("execute if no exception")
finally:
  print("always executed")

Output:-
Something went wrong
always executed

# Example_4 for try except else finally
try:
  print("x")
except:
  print("Something went wrong")
else:
  print("execute if no exception")
```

```
finally:
  print("always executed")

Output:-
x
execute if no exception
always executed
```

Handling Error

```
# ZeroDivisionError
try:
    result = 10 / 0
    print(result)
except ZeroDivisionError:
    print("You can't divide by zero!")      # Output:- You can't divide by zero!


# Index Error
try:
    even_numbers = [2,4,6,8]
    print(even_numbers[5])
except IndexError:
    print("Index Out of Bound.")    # index is not found


# Name Error
try:
    print(my_variable)
except NameError as e:
    print(f"A NameError occurred: {e}") # A NameError occurred: name 'my_variable'
is not defined


# TypeError
try:
    result = "Hello" + 5  # Attempting to add a string and an integer
except TypeError as e:
    print(f"A TypeError occurred: {e}") # A TypeError occurred: can only
concatenate str (not "int") to str




# Catch Multiple Exceptions in Python
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print(result)
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

```
except ValueError:
    print("Invalid input. Please enter a valid number.")
except Exception as e:
    print(f"An error occurred: {e}")
```

- Python try with else clause

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)

Output:-
Enter a number: 1
Not an even number!

Output:-
Enter a number: 4
0.25

Output:-
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

- Python try...finally

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")

Output:-
```

```
Error: Denominator cannot be 0.
This is finally block.
```

Errors and Exceptions in Python

1. Syntax Errors:

```
print("Hello, World!"  # Missing closing parenthesis

#
a = 10000
if a > 2999
    print("Eligible")

syntax error because there is a missing colon (:) after the if statement.
```

2. Logical Errors:

MemoryError

```
try:
    # Attempting to create a very large list
    large_list = [0] * (10**10)  # Trying to create a list with 10 billion
elements
except MemoryError as e:
    print(f"A MemoryError occurred: {e}")
```

DataTime Function

Attributes of datetime Module

```
import datetime

print(dir(datetime))

Output:-
['MAXYEAR', 'MINYEAR', 'UTC', '__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__', 'date',
'datetime', 'datetime_CAPI', 'time', 'timedelta', 'timezone', 'tzinfo']
```

Get Current Date and Time

```
import datetime
# get the current date and time
now = datetime.datetime.now()
print(now)
print("year", now.year)
print("month", now.month)
print("day", now.day)
print("hour", now.hour)
print("minutes", now.minute)
print("second", now.second)

Output:- 2024-12-25 10:16:05.190706
year 2024
month 12
day 25
hour 10
minutes 53
second 3

(OR)
from datetime import datetime

# returns current date and time
now = datetime.now()
print("now = ", now)

Output:- now =  2024-12-25 10:35:43.891754
```

Get Current Date

```
import datetime
# get current date
current_date = datetime.date.today()
print(current_date)

(OR)
# Import date class from datetime module
from datetime import date
# Returns the current local date
today = date.today()
print("Today date is: ", today)

Output:- 2024-12-25
```

Format change data and time

```
from datetime import datetime

# returns current date and time
now = datetime.now().strftime("%Y/%m/%d")    # change the format
print("now = ", now)
```

Do I Get a Specific Date in Python?

```
from datetime import date

# Create a specific date
specific_date = date(2024, 12, 25)
print(specific_date)  # Output: 2024-12-25

# with time parameters as well
a = datetime(2022, 10, 22, 6, 2, 32, 5456)
print(a)  # Output:- 2022-10-22 06:02:32.005456
```

Ques. What is File Handling in Python?

- Suppose you are working on a file saved on your personal computer. If you want to perform any operation on that file like opening it, updating it or any other operation on that, all that comes under File handling.
- File handling in Python involves interacting with files on your computer to read data from them or write data to them. Python provides several built-in functions and methods for creating, opening, reading, writing, and closing files.
- Types Of File in Python
- Binary file
    - **Document files:** .pdf, .doc, .xls etc.
    - **Image files:** .png, .jpg, .gif, .bmp etc.
    - **Video files:** .mp4, .3gp, .mkv, .avi etc.
    - **Audio files:** .mp3, .wav, .mka, .aac etc.
    - **Database files:** .mdb, .accde, .frm, .sqlite etc.
    - **Archive files:** .zip, .rar, .iso, .7z etc.
    - **Executable files:** .exe, .dll, .class etc.
- Text file
    - **Web standards:** html, XML, CSS, JSON etc.
    - Source code: c, app, js, py, java etc.
    - Documents: txt, tex, RTF etc.
    - Tabular data: csv, tsv etc
    - Configuration: ini, cfg, reg etc

Modes	Description
r	Opens a file in read-only mode. The pointer of the file is at the beginning of the file. This is also the default mode.
rb	Opens a file for reading only in binary format.
r+	Opens the file for reading and writing. The pointer is at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

Modes	Description
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
x	open for exclusive creation, failing if the file already exists
+	open file for updating (reading and writing)
b	Opens the file in binary mode
t	Opens the file in text mode (default)

Opening a File in Python

- This function takes two arguments. First is the filename along with its complete path, and the other is access mode. This function returns a file object.
- To perform any file operation, the first step is to open the file. Python's built-in open() function is used to open files in various modes, such as reading, writing, and appending. The syntax for opening a file in Python is –

```
file = open("filename", "mode")
```

```
#example 1
print("hello")
# file1 =
open("C:\Users\mohits4\Desktop\mohit\study\python\python\1_python\code\example.txt
", "r")
file1 = open("example.txt", "r")
print(file1.read())

#example 2
print("hello")
# file1 =
open("C:\Users\mohits4\Desktop\mohit\study\python\python\1_python\code\example.txt
", "r")
file1 = open("example.txt", "a")
file1.write("Now the file has more content!")
file1.close()
file1 = open("example.txt", "r")
print(file1.read())
```

How to read file in python?

- Open a File on the Server:- The **open()** function returns a file object, which has a **read() method** for reading the content of the file.

```
file = open("file_name.txt", "r")  # Opens the file in read mode
content = file.read()  # Read the content of the file
print(content)
file.close()  # Close the file when done
```

- By default the read() method returns the whole text, but you can also specify how many characters you want to return.

```
f = open("demofile.txt", "r")
print(f.read(5))

Output:- Hello
```

- The **readline() function** helps you read a **single** line from the file.

```
file_obj = open("example.txt", "r", encoding="utf-8")
print(file_obj.readline())
file_obj.close()

Output:-
Hello World
```

- As the name suggests, the **readlines() method** reads **all the lines** in the file and returns them properly separated in a list.

```
file_obj = open("example.txt", "r", encoding="utf-8")
print(file_obj.readlines())
file_obj.close()

Output:-
This is a new file.
We have now learnt all read functions.
We are trying a new method
to loop over files.
```

- **Close Files:-** It is a good practice to always close the file when you are done with it.

```
file = open("demofile.txt", "r")
print(f.readline())
file.close()
```

How to Write a File in Python

- We're creating it using the **w** mode. Once we open the new file, it's obviously empty. We're then going to write content into it.
- Write - will overwrite any existing content and Create a new file if it does not exist.

```
file_obj = open("writing.txt", "w")
```

- **a** Append - will append to the end of the file

```
file = open("writing.txt", "a")
file.write("This way, I will preserve the existing contents in the file")
print(file.read())
file.close()
```

using with statement?

- The method shown in the above section is not entirely safe. If some exception occurs while opening the file, then the code will exit without closing the file.

```
# Opening file in read mode and printing the contents of the file.
with open("test.txt", mode='r') as f:
    data = f.readlines() #This reads all the lines from the file in a list.
    print(data) #This will print the content of the Hello World file!

# Opening a file in write mode.
with open("test.txt", mode='w') as f:
    f.write("Data after write operation")
# Opening file in read mode to check the contents.
with open("test.txt", mode='r') as f:
    data = f.readlines() # this reads all the lines from the file in a list.
    print(data) #this will print the overwritten content of the file that is
"Data after write operation"

# Opening a file in append mode and appending data to the file.
with open("test.txt", "a") as f:
    f.write(" Appending new data to the file")
# Opening file in read mode to check the contents.
with open("test.txt", mode='r') as f:
    data = f.readlines() #This reads all the lines from the file in a list.
```

```
    print(data) #this will print the existing content of file plus the appended
content
```

Ques. How do you remove a file from a folder in python?

????p

Ques. Program to Delete all files with a specific extension?

```
import os
from os import listdir
my_path = 'C:\Python Pool\Test\'
for file_name in listdir(my_path):
    if file_name.endswith('.txt'):
        os.remove(my_path + file_name)
```

What is the Python “with” statement designed for?

Creates a file

```
# Using open() with Write Mode ('w'):- This method creates a new file or truncates
an existing file.
with open('example.txt', 'w') as file:
    file.write("This is a new file created in write mode.")

# Using open() with Append Mode ('a'):- This method creates a new file if it
doesn't exist and appends content to it if it does.
with open('example.txt', 'a') as file:
    file.write("\nThis line is added to the existing file.")

# Using open() with Exclusive Creation Mode ('x')
try:
    with open('example.txt', 'x') as file:
        file.write("This file is created using exclusive mode.")
except FileExistsError:
    print("File already exists.")
```

Read files

```
# using read() method
f = open("example.txt", "r")
print(f.read())
f.close()


# Using the with statement
```

```
with open("demofile.txt") as f:
  print(f.read())

# using loop
with open("demofile.txt") as f:
  for x in f:
    print(x)


# Return the 5 first characters of the file:
with open("demofile.txt") as f:
  print(f.read(5))


# Using open() with Read Mode ('r')
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

# Read Lines
# You can return one line by using the readline() method:
with open("example.txt") as f:
  print(f.readline())

# By calling readline() two times, you can read the two first lines:
with open("example.txt") as f:
  print(f.readline())
  print(f.readline())

# Reading Line by Line:- You can read a file line by line using a loop.
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())  # Use strip() to remove newline characters


# Using readline():- The readline() method reads one line at a time. You can call
it multiple times to read subsequent lines.
with open('example.txt', 'r') as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())
```

write a file

```
# Using open() with Write Mode ('w'):- This method creates a new file or truncates
an existing file.
with open('example.txt', 'w') as file:
    file.write("This is a new file created in write mode.")

# Using open() with Append Mode ('a'):- This method creates a new file if it
```

```
doesn't exist and appends content to it if it does.
with open('example.txt', 'a') as file:
    file.write("\nThis line is added to the existing file.")

# Using open() with Exclusive Creation Mode ('x')
try:
    with open('example.txt', 'x') as file:
        file.write("This file is created using exclusive mode.")
except FileExistsError:
    print("File already exists.")

# Writing Multiple Lines:- You can write multiple lines to a file using the
writelines() method.
lines = ["First line.\n", "Second line.\n", "Third line.\n"]
with open('example.txt', 'w') as file:
    file.writelines(lines)
```

Delete file

```
# os.remove() function
import os

file_name = 'example.txt'
try:
    os.remove(file_name)
    print(f"{file_name} has been deleted.")
except FileNotFoundError:
    print(f"{file_name} does not exist.")
except PermissionError:
    print(f"Permission denied to delete {file_name}.")


# Using os.unlink()
import os

file_name = 'example.txt'
try:
    os.unlink(file_name)
    print(f"{file_name} has been deleted.")
except FileNotFoundError:
    print(f"{file_name} does not exist.")
except PermissionError:
    print(f"Permission denied to delete {file_name}.")

# Using os.rmdir() for Directories:- If you need to delete an empty directory, you
can use os.rmdir().
# Note that this will only work if the directory is empty.
import os

directory_name = 'empty_directory'
try:
```

```
    os.rmdir(directory_name)
    print(f"{directory_name} has been deleted.")
except FileNotFoundError:
    print(f"{directory_name} does not exist.")
except OSError:
    print(f"{directory_name} is not empty or cannot be removed.")

# Using shutil.rmtree() for Non-Empty Directories
# If you want to delete a directory and all its contents (files and
subdirectories), you can use the shutil module.
# to delete non-empty directories.
import shutil

directory_name = 'non_empty_directory'
try:
    shutil.rmtree(directory_name)
    print(f"{directory_name} and all its contents have been deleted.")
except FileNotFoundError:
    print(f"{directory_name} does not exist.")
except PermissionError:
    print(f"Permission denied to delete {directory_name}.")
```

Ques. What is an Iterable?

- Lists, tuples, dictionaries, Strings, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
- An Iterable is an object that implements the **iter**() method and returns an iterator object or an object that implements **getitem**() method (and should raise an IndexError when indices are exhausted).

What is Iterators?

- iterator is an object, its print the value one by one.
- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods **iter**() and **next**().

```
# Example1
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))

Output:-
apple
banana
cherry

# Example 2
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))

Output:-
b
a
n
a
n
a

# Looping Through an Iterator
# The for loop actually creates an iterator object and executes the next() method
for each loop
mytuple = ("apple", "banana", "cherry")
```

```
for x in mytuple:
  print(x)

Output:-
apple
banana
cherry
```

Ques. Difference between Iterators and iterable?

- Every iterator is also an iterable, but not every iterable is an iterator.

Iterable	Iterator
An Iterable is basically an object that any user can iterate over.	An Iterator is also an object that helps a user in iterating over another object (that is iterable).
We can generate an iterator when we pass the object to the iter() method.	We use the **next**() method for iterating. This method helps iterators return the next item available from the object.
Every iterator is basically iterable.	Not every iterable is an iterator.

Ques. Difference between generator and iterators in python?

Iterator	Generator
Class is used to implement an iterator	Function is used to implement a generator.
Local Variables aren't used here.	All the local variables before the yield function are stored.
Iterators are used mostly to iterate or convert other objects to an iterator using iter() function.	Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop
Iterator uses iter() and next() functions	Generator uses yield keyword
Every iterator is not a generator	Every generator is an iterator

Logging

create the file

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("app.log"),  # Output logs to a file
        logging.StreamHandler()          # Output logs to the console
    ]
)

logger = logging.getLogger(__name__)

@router.get("/user-list-with-all-data/")
def get_user_list(db: Session = Depends(get_db)):
    logger.info("Root endpoint accessed11")
```

create the file under the log foler

```
# Define the log folder and file name
log_folder = "logs"
log_file_name = "app.log"

# Create the log folder if it doesn't exist
if not os.path.exists(log_folder):
    os.makedirs(log_folder)

# Create the full path to the log file
log_file_path = os.path.join(log_folder, log_file_name)

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler(log_file_path),  # Output logs to the specified file
        logging.StreamHandler()              # Output logs to the console
    ]
)

logger = logging.getLogger(__name__)
```

create the file under the log foler with date

```
# Generate the current date
current_date = datetime.now().strftime("%d_%m_%Y")


log_folder = "logs"
log_file_name = f"app_{current_date}.log"
```

log calling in function

```
# with all data
@router.get("/user-list-with-all-data/")
def get_user_list(db: Session = Depends(get_db)):
    logger.info("Root endpoint accessed11")
```

log config in saprate file

```
project/
│
├── logs/
│   └── app_25_12_2024.log
│
├── logger_config.py
├── main.py

create the logger_config.py file
and put the code

# and call main.py file
from logger_config import logger

def main():
    # Example log messages
    logger.info("This is an info message from main.py")
    logger.error("This is an error message from main.py")
```

saprate file for log and call in main.py file.

```
# Create the logger_config.php and put the code
#-----------------------------------------------
import os
import logging
from datetime import datetime

def configure_logging():
    # Generate the current date
    current_date = datetime.now().strftime("%d_%m_%Y")
```

```
    # Define the log folder and file name
    log_folder = "logs"
    log_file_name = f"app_{current_date}.log"

    # Create the log folder if it doesn't exist
    if not os.path.exists(log_folder):
        os.makedirs(log_folder)

    # Create the full path to the log file
    log_file_path = os.path.join(log_folder, log_file_name)

    # Configure logging
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler(log_file_path),  # Output logs to the specified
file
            logging.StreamHandler()              # Output logs to the console
        ]
    )

# Call the function to configure logging
configure_logging()

# Get the logger
logger = logging.getLogger(__name__)

# call in main.py/route file
#_____________________________

from logger_config import logger

@router.get("/user-list-with-all-data/")
def get_user_list(db: Session = Depends(get_db)):
    logger.info("Root endpoint accessed11")
```

using package

- refrence:- https://loguru.readthedocs.io/en/stable/overview.html
- install package

```
pip install loguru
```

Math Module

- Python has a built-in module that you can use for mathematical tasks.

Math Methods

- **Arithmetic operations:** math.ceil(), math.floor(), math.fabs(), math.fmod(), math.pow(), math.sqrt(), math.trunc().
- **Trigonometric functions:** math.sin(), math.cos(), math.tan(), math.asin(), math.acos(), math.atan(), math.atan2(), math.hypot().
- **Logarithmic and exponential functions:** math.exp(), math.log(), math.log10(), math.log2().
- **Angular conversion:** math.degrees(), math.radians().
- **Constants:** math.pi, math.e, math.tau, math.inf, math.nan.

Example

```
result = math.sqrt(25)  # Calculates the square root of 25
print(result)  # Output: 5.0
```

math.floor() Method

- The math.floor() method rounds a number DOWN to the nearest integer, if necessary, and returns the result.

```
#Import math library
import math

# Round numbers down to the nearest integer
print(math.floor(0.6))          # 0
print(math.floor(1.4))          # 1
print(math.floor(5.3))          # 5
print(math.floor(-5.3))         # -6
print(math.floor(22.6))         # 22
print(math.floor(10.0))         # 10
```

math.ceil() Method

- The math.ceil() method rounds a number UP to the nearest integer, if necessary, and returns the result.

```
#Import math library
import math

#Round a number upward to its nearest integer
print(math.ceil(1.4))           # 2
print(math.ceil(5.3))           # 6
print(math.ceil(-5.3))          # -5
```

```
print(math.ceil(22.6))          # 23
print(math.ceil(10.0))          # 10
```

Multithreading?

- In Python Multithreading is a technique that allows a program to run multiple tasks simultaneously.
- Multithreading is a technique in programing that allows multiple threads of execution to run concurrently within a single process.
- In Python, We can use the threading module to implement multithreading.

Benefits:-

- It's a powerful tool that can improve application performance and responsiveness.

Steps Multithreading in Python

- **Step 1:** Import Module

```
import threading
```

- **Step 2:** Create a Thread
  - we create an object of the Thread class. It takes the **'target'** and **'args'** as the parameters. The target is the **function** to be executed by the thread whereas the args is the **arguments** to be passed to the target function.

```
t1 = threading.Thread(target, args)
t2 = threading.Thread(target, args)
```

- **Step 3:** Start a Thread
  - To start a thread, we use the start() method of the Thread class.

```
t1.start()
t2.start()
```

- **Step 4:** End the thread Execution
  - Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop the execution of the current program until a thread is complete, we use the join() method.

```
t1.join()
t2.join()
```

Example:-

```

```

```python
# Normal calling
import threading
from time import sleep, perf_counter

def fun1():
    start_time = perf_counter()
    print("start ")
    fun2()
    fun3()
    print("end")
    end_time = perf_counter()
    print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')


def fun2():
    sleep(10)
    print("fun2 completed")

def fun3():
    sleep(4)
    print("fun3 completed")


fun1()

# throwh threading calling
from threading import Thread
from time import sleep, perf_counter

def fun1():
    start_time = perf_counter()
    print("start ")
    t1 = Thread(target=fun2)
    t2 = Thread(target=fun3)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print("end")
    end_time = perf_counter()
    print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')


def fun2():
    sleep(10)
    print("fun2 completed")

def fun3():
    sleep(4)
    print("fun3 completed")


fun1()
```

```
import threading

def print_cube(num):
    print("Cube: {}" .format(num * num * num))

def print_square(num):
    print("Square: {}" .format(num * num))

# print_cube(10)

t1 = threading.Thread(target=print_square, args=(10,))
t2 = threading.Thread(target=print_cube, args=(10,))

t1.start()
t2.start()

t1.join()
t2.join()
```

Python Closures

- Python closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.
- function ke ander ke function ko nested function kahte hai.

```
def functionA():
   print ("Outer function")
   def functionB():
      print ("Inner function")
   functionB()

functionA()

Output:
Outer function
Inner function
```

```
def functionA(name):
   print ("Outer function")
   def functionB():
      print ("Inner function")
      print ("Hi {}".format(name))
   functionB()

functionA("Python")

Output:-
Outer function
Inner function
Hi Python
```

```
def greet(name):
    # inner function
    def display_name():
        print("Hi", name)

    # call inner function
    display_name()

# call outer function
greet("John")

# Output: Hi John
```

↑ Back to Top