Module - Python Fundamentals

1. Introduction to Python Theory:

Que.1 Introduction to Python and its Features (simple, high-level, interpreted language).

➔ **Python** is a high-level, interpreted programming language created by Guido van Rossum in the late 1980s. It was designed with simplicity and readability in mind, making it an excellent choice for both beginners and professionals. Python's unique syntax emphasizes code readability, allowing developers to express concepts in fewer lines of code than other languages.Python has gained immense popularity and has become one of the most widely used programming languages in the world.

- **High-Level Language:** Python is a high-level programming language, which means it abstracts many low-level details like memory management and hardware interactions. This makes it easier to write and read Python code compared to lower-level languages like C or assembly.
- **Interpreted Language:** Python is an interpreted language, which means that you don't need to compile your code before running it. Instead, an interpreter reads and executes the code line by line, making development and debugging faster.

Que.2 History and evolution of Python.

➔ **Python** is a widely used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed to emphasize code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Que.3 Advantages of using Python over other programming languages.

➔ 1. Python is Simple to Learn and Use
2. Perfect for Rapid Deployment
3. Python has High-Performance
4. Strong Support for Data Analysis and Scientific Computing
5. User-Friendly Built-in Data Types
6. Python Needs Less Coding
7. Python Has OS Portability
8. Python is Flexible in Deployment

9. Support from Renowned Corporate Sponsors
10. Hundreds of Community Driven Python Libraries and Frameworks

Que.4 Installing Python and setting up the development environment (VS Code).

➔ **Virtual Studio Code (VSCode)** is a perfect Integrated Development Environment for Python. It is simple and comes with built-in features that enhance the development experience. VSCode Python extensions come with powerful features like syntax autocomplete, linting, debugging, unit testing, GitOps, virtual environments, notebooks, editing tools, and the ability to customize the editor.

Key Features:

1. Command Palette to access all commands by typing keywords.
2. Fully customizable keyboard shortcuts.
3. Jupyter extension for data science. Run Jupyter notebook within the IDE.
4. Auto linting and formatting.
5. Debugging and Testing.
6. Git integration.
7. Custom code snippets..
8. Enhanced editing tools. Multi cursor selection, column selection, outline view, side-by-side preview, and search and modify.

In this tutorial, we will start by installing Python and VSCode, then run a Python script in VSCode. After that, we will customize the editor to enhance the Python development experience by installing essential extensions and learning about built-in features. In the end, we will learn about Python productivity hacks.

2. Programming Style:

Que. 1 Understanding Python's PEP 8 guidelines.

➔ **The PEP** 8 Style Guide is a document that provides guidelines for writing Python code. It was created by Guido van Rossum, the creator of Python, and is maintained by the Python Software Foundation. The purpose of the guide is to provide a set of coding conventions that will make it easier for developers to read, write, and maintain Python code.

PEP 8 important :

➔ Following the guidelines outlined in the PEP 8 Style Guide is important because it makes your code more readable and easier to maintain. When multiple developers work on a project, it's important that the code follows a consistent style. This makes it easier for developers to understand the code and to make changes to it. Additionally, when code is consistent, it's easier to spot errors and bugs.

Basic conventions :

➔ The PEP 8 Style Guide provides guidelines for naming conventions, indentation, and spacing. Here are some of the most important basic conventions:

1. Naming Conventions
2. Indentation
3. Spacing
4. Whitespace
5. Comments
6. Funcation And Classes
7. Imports
8. Conclusion

Que. 2 Indentation, comments, and naming conventions in Python.

➔ A Python statement is an instruction that the Python interpreter can execute. There are different types of statements in Python language as Assignment statements, Conditional statements, Looping statements, etc. The token character NEWLINE is used to end a statement in Python. It signifies that each line of a Python script contains a statement. These all help the user to get the required output.

The different types of Python statements are listed below:

Multi-Line Statements

- Python Conditional and Loop Statements
  - Python If-else
  - Python for loop
  - Python while loop
  - Python try-except
  - Python with statement

- Python Expression statements
    - Python pass statement
    - Python del statement
    - Python return statement
    - Python import statement
    - Python continue and
    - Python break statement

➔ A comment can be written on a single line, next to the corresponding line of code, or in a block of multiple lines. Here, we will try to understand examples of comment in Python one by one:

➔ Single-line comment in Python

Python single-line comment starts with a hash symbol (#) with no white spaces and lasts till the end of the line. If the comment exceeds one line then put a hashtag on the next line and continue the comment. Python's single-line comments are proved useful for supplying short explanations for variables, function declarations, and expressions. See the following code snippet demonstrating single line comment

➔ Multiline comment in Python

Use a hash (**#)** for each extra line to create a multiline comment. In fact, Python multiline comments are not supported by Python's syntax. Additionally, we can use Python multi-line comments by using multiline strings. It is a piece of text enclosed in a delimiter (“””) on each end of the comment. Again there should be no white space between delimiter (“””). They are useful when the comment text does not fit into one line; therefore need to span across lines. Python Multi-line comments or paragraphs serve as documentation for others reading your code. See the following code snippet demonstrating a multi-line comment:

Que. 2 Writing readable and maintainable code.

➔ In this article, we will discuss the importance of writing readable and maintainable code in Python. Python is a popular language used for web development, data analysis, machine learning, and more. However, as codebases grow, it becomes increasingly important to write code that is easy to read and maintain.

Readable and maintainable code is essential for several reasons:

- ➔ Reduces Bugs and Errors: Well-written code is less likely to contain bugs and errors, which saves time and resources in the long run.
- ➔ Improves Collaboration: When code is readable and maintainable, multiple developers can collaborate on a project without getting confused or lost.
- ➔ Increases Productivity: Code that is easy to understand and modify allows developers to work more efficiently.

1. Use Meaningful Variable Names

Variable names should be descriptive and indicate what the variable represents. For example:

```
total_price = 10.99
```

Instead of:

```
x = 10.99
```

2. Follow PEP 8 Guidelines

PEP 8 is a style guide for Python code that recommends best practices for indentation, spacing, and naming conventions. Follow these guidelines to ensure your code is readable and consistent.

3. Use Comments and Docstrings

Comments and docstrings provide context and explain the purpose of code. Use them to describe complex logic, explain why certain decisions were made, and document functions and classes.

4. Keep Functions Short and Focused

Functions should perform a single task and be short in length. This makes it easier to understand and maintain code.

5. Avoid Global Variables

Global variables can lead to tight coupling and make code harder to understand. Instead, pass variables as arguments to functions or use a dependency injection pattern.

6. Use Type Hints and Docstrings

Type hints and docstrings provide additional context and make it easier for others to understand your code.

By following these best practices, you can write Python code that is readable, maintainable, and efficient. Remember, clean code is not just about following rules – it's about writing code that is easy to understand and modify.

3. Core Python Concepts

Que.1 Understanding data types: integers, floats, strings, lists, tuples,Dictionaries, sets.

➔ In programming, data types are classifications that specify which type of value a variable can hold. Data types define the set of operations that can be performed on the data and the way data is stored in memory. Understanding data types is crucial because it helps ensure that the data is used appropriately and that the code behaves as expected.

➔ Python is a dynamically typed language, which means you don't need to specify the data type of a variable explicitly. Python infers the data type based on the value assigned to the variable. This dynamic typing allows for flexibility but requires careful handling of data types to avoid unexpected behavior.

Common Python Data Types

Let's explore some of the most common data types in Python:

1. **Integers (int):** Integers represent whole numbers, both positive and negative. They have no fractional part.

X=5

Y=-1

2. Floating-Point Numbers (float): Floating-point numbers, or floats, represent real numbers with decimal points. They can represent both integers and fractions.

pi=3.14159
temperature = -22.5

3. Strings (str): Strings represent sequences of characters and are enclosed in single (‘) or double (“) quotes.

Name="Alice"
greeting = 'Hello, Python!'

4. Booleans (bool): Booleans represent binary values — True or False. They are often used in conditional statements and comparisons.

is_true=True
is_false = False

5. Lists: Lists are ordered collections of items that can be of any data type. They are defined using square brackets [].

fruits=["apple","banana","cherry"]
numbers = [1, 2, 3, 4, 5]

6. Tuples: Tuples are similar to lists but are immutable, meaning their values cannot be changed after creation. They are defined using parentheses ().

coordinates=(3.5,2.7)
person_info = ("Alice", 25, "New York")

7. Dictionaries (dict): Dictionaries are collections of key-value pairs, used to store and retrieve data efficiently. They are defined using curly braces {}.

person={"name":"Bob","age":30,"city":"Chicago"}
website = {"name":"wordpediax","category":"Tech Blogging", "Platform": "Google"}

8. Sets: Sets are unordered collections of unique elements. They are defined using curly braces {} or the set() constructor.

unique_numbers={1,2,3,4,5}
letters = set("python")

Que.2 Python variables and memory allocation.

➔ Memory allocation can be defined as allocating a block of space in the computer memory to a program. In Python memory allocation and deallocation method is automatic as the Python developers created a garbage collector for Python so that the user does not have to do manual garbage collection.

Garbage Collection :

Garbage collection is a process in which the interpreter frees up the memory when not in use to make it available for other objects.

Assume a case where no reference is pointing to an object in memory i.e. it is not in use so, the virtual machine has a garbage collector that automatically deletes that object from the heap memory

Reference Counting :

Reference counting works by counting the number of times an object is referenced by other objects in the system. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero, the object is deallocated.

For example, Let's suppose there are two or more variables that have the same value, so, what Python virtual machine does is, rather than creating another object of the same value in the private heap, it actually makes the second variable point to that originally existing value in the private heap. Therefore, in the case of classes, having a number of references may occupy a large amount of space in the memory, in such a case referencing counting is highly beneficial to preserve the memory to be available for other objects

Que.3 Python operators: arithmetic, comparison, logical, bitwise.

➔ Logical Operators in Python are used to perform logical operations on the values of variables. The value is either true or false. We can figure out the conditions by the result of the truth values. There are mainly three types of logical operators in python: logical

AND, logical OR and logical NOT. Operators are represented by keywords or special characters.

Arithmetic Operators

Arithmetic Operators perform various arithmetic calculations like addition, subtraction, multiplication, division, %modulus, exponent, etc. There are various methods for arithmetic calculation in Python like you can use the eval function, declare variable & calculate, or call functions.

Example: For arithmetic operators we will take simple example of addition where we will add two-digit 4+5=9

```
x= 4

y= 5

print(x + y)
```

Similarly, you can use other arithmetic operators like for multiplication(*), division (/), substraction (-), etc.

Comparison Operators

Comparison Operators In Python compares the values on either side of the operand and determines the relation between them. It is also referred to as relational operators. Various comparison operators in python are (==, != , <>, >,<=, etc.)

Example: For comparison operators we will compare the value of x to the value of y and print the result in true or false. Here in example, our value of x = 4 which is smaller than y = 5, so when we print the value as x>y, it actually compares the value of x to y and since it is not correct, it returns false

```
x = 4

y = 5

print(('x > y  is',x>y))
```

Python Assignment Operators

Assignment Operators in Python are used for assigning the value of the right operand to the left operand. Various assignment operators used in Python are (+=, – = , *=, /= , etc.).

Example: Python assignment operators is simply to assign the value, for example

```
num1 = 4

num2 = 5

print(("Line 1 - Value of num1 : ", num1))

print(("Line 2 - Value of num2 : ", num2))
```

Logical Operators or Bitwise Operators

Logical operators in Python are used for conditional statements are true or false. Logical operators in Python are AND, OR and NOT. For logical operators following condition are applied.

- For AND operator – It returns TRUE if both the operands (right side and left side) are true
- For OR operator- It returns TRUE if either of the operand (right side or left side) is true
- For NOT operator- returns TRUE if operand is false

Example: Here in example we get true or false based on the value of a and b

```
a = True

b = False

print(('a and b is',a and b))

print(('a or b is',a or b))

print(('not a is',not a))
```

4. Conditional Statements

Que 1. Introduction to conditional statements: if, else, elif.

➔ Conditional statements in programming allow the execution of different pieces of code based on whether certain conditions are true or false. Here are five common types of conditional statements:

1. If Conditional Statement:

The if statement is the most basic form of conditional statement. It checks if a condition is true. If it is, the program executes a block of code.

Syntax of If Conditional Statement:

```
if (condition):
 {
   // code to execute if condition is true
 }
```

2. If-Else Conditional Statement:

The if-else statement extends the if statement by adding an else clause. If the condition is false, the program executes the code in the else block.

Syntax of If-Else Conditional Statement:

```
if (condition) :
{
   // code to execute if condition is true
}
else :
{
   // code to execute if condition is false
}
```

3. if-Else if Conditional Statement:

The if-else if statement allows for multiple conditions to be checked in sequence. If the if condition is false, the program checks the next else if condition, and so on.

Syntax of If-Else if Conditional Statement:

```
if (condition1) :
{
   // code to execute if condition1 is true
}
 else if (condition2) :
```

```
{
   // code to execute if condition2 is true
} else :

{
   // code to execute if all conditions are false
}
```

Que 2. Nested if-else conditions.

➔ There comes a situation in real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we choose when to execute the next block of code. This is done with the help of a Nested if statement.

➔ In Python a Nested if statement, we can have an if…elif…else statement inside another if…elif…else statement. This is called nesting in computer programming. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so it must be avoided if we can.

5. Looping (For, While)

Que 1. Introduction to for and while loops.

➔ Python programming language provides two types of Python loopshecking time. In this article, we will look at Python loops and understand their working with the help of examp – For loop and While loop to handle looping requirements. Loops in Python provides three ways for executing the loops.

➔ While all the ways provide similar basic functionality, they differ in their syntax and condition-checking time. In this article, we will look at Python loops and understand their working with the help of examples.

While Loop in Python

In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

Python While Loop Syntax:

While expression:
    statement(s)

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

For Loop in Python

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is "for in" loop which is similar to foreach loop in other languages. Let us learn how to use for loops in Python for sequential traversals with examples.

For Loop Syntax:

For iterator_var insequen:
    statements(s)

It can be used to iterate over a range and iterators.

Que 2. How loops work in Python.

➔ The main reason why we need looping in programs is to make complex problems simpler as sometimes we won't be writing the whole program to go through each query if the functionality is same. For example, if we want to print the numbers from 1 to 10, we could use iterations inside a loop; in that way we can only write the code once and repeat the functionality as much as we want through looping.

➔ The main advantages of using loops in python is that it saves your time, you don't have to write the same code every time to perform the same functionality; also loops make your code look clean.

- **While Loop**
  A while loop in python continues to execute the code as long as the given condition is true, if it becomes false, then the loop stops.

- **For Loops**
  For loop is a python loop which repeats a group of statements for a given number of times. It is always a good practice to use the for loop when we know the number of iterations.

- **Nested Loops**
  Python supports using the nested loop that is a loop within a loop. The following example is an explanation of how a nested loop works.

Que 3. Using loops with collections (lists, tuples, etc.).

➔ Iterating Over Lists

Lists are ordered, mutable collections of elements. You can use for loops or while loops to iterate over them.

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

➔ Iterating Over Tuples

Tuples are immutable collections. The iteration process is similar to lists.

```
coordinates = (10, 20, 30)
for coord in coordinates:
    print(coord)
```

➔ Iterating Over Sets

Sets are unordered collections of unique elements. Iteration is done directly over elements, but the order is not guaranteed.

```
unique_numbers = {1, 2, 3, 4}
for num in unique_numbers:
    print(num)
```

6. Generators and Iterators

Que. 1 Understanding how generators work in Python.

➔ In the world of Python programming, generators provide a unique and efficient way to work with sequences of data without the need to store them in memory. Unlike a regular function that returns a single value, a generator yields a sequence of values, one at a time, allowing for iteration over the sequence. The beauty of Python generators is that they allow for lazy evaluation – meaning they only produce values as they're needed, rather than calculating them upfront. This leads to significant memory savings and potential performance improvements, especially when dealing with large datasets.

➔ One might wonder how a generator differs from a list comprehension or a for-loop. Ponder the classic example of generating the Fibonacci sequence:

```
def fibonacci(n):
a, b = 0, 1
for _ in range(n):
yield a
a, b = b, a + b
```

➔ By using the *yield* statement instead of returning a list, we've created a generator. At each iteration, it gives us the next Fibonacci number without having to store all the previous numbers in memory. This would not be possible with a regular function or list comprehension which would need to store the entire sequence before even starting to iterate.

➔ A typical workflow with generators involves defining a generator function and then consuming its results using a loop or converting it into another sequence type. We will dive deeper into the creation and usage of generators in the following sections of the article.

➔ Generators are particularly handy when dealing with data streams, infinite sequences, or when applying transformations to elements of a collection one at a time. You can also leverage generator expressions for simple use cases, which look very much like list comprehensions but use parentheses instead of brackets:

```
gen_expr = (x ** 2 for x in range(10))
for value in gen_expr:
print(value)
```

➔ As we proceed through this article, we'll explore how to create and work with generator functions and objects. We'll also discuss advanced usage scenarios and further benefits that make Python generators an invaluable tool in any developer's toolkit.

➔ Creating generators in Python

➔ To create a generator in Python, you use either a generator function or a generator expression. A generator function is similar to a regular function, but it uses the *yield* keyword instead of *return* to provide a result to its caller without stopping its execution. This means that you can call a generator function multiple times and it will yield a new value on each call.

```
def count_up_to(max):
count = 1
while count
```

➔ In this example, *count_up_to* is a generator function that yields an increasing sequence of numbers up to a specified maximum. When this function is called, it doesn't run to completion like a normal function. Instead, it pauses each time it reaches the *yield* statement and resumes from this point when next called.

➔ A generator expression is another way to create a generator and it looks very similar to list comprehension. The major difference is that list comprehensions produce the entire list in memory at the same time, whereas generator expressions produce items one at a time and are enclosed in parentheses instead of square brackets.

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

➔ This generator expression creates an generator object that generates square numbers. Unlike a list comprehension, it will only compute the squares on demand.

➔ Both methods of creating generators lead to a **memory-efficient** way to work with potentially large collections of items, as only one item needs to be in memory at a time. You can convert the generator into a list if you want to collect all the produced items at once:

```
squares = list(gen_exp) # [0, 1, 4, 9, 16]
```

➔ However, the true benefit of a generator comes from their lazy evaluation allowing for operations on large or even infinite sequences without the overhead of storing the entire sequence in memory at any one time.

➔ Working with generator functions

➔ When working with generator functions in Python, it's essential to understand how the *yield* statement functions. When a generator function calls *yield*, the state of the function is frozen, and the value to the right of the yield expression is returned to the caller. Upon resumption, the function continues execution right after the yield statement from where it left off. That's a stark contrast from a regular function where invoking a return statement halts execution and exits the function completely.

```
def reverse_string(my_string):
length = len(my_string)
for i in range(length - 1, -1, -1):
yield my_string[i]
# Create a generator object
reversed_gen = reverse_string("Python")
# Iterate through the generator object to consume the values
for char in reversed_gen:
print(char)
```

➔ This reverse_string generator function yields characters of a string in reverse order, one by one. Instead of creating and returning a reversed string all at the same time, it allows for iteration over each character as needed. This on-demand calculation exemplifies how generators provide lazy evaluation.

➔ Another aspect of generator functions involves sending values back into the generator after each yield. This advanced feature is achieved using the *generator.send()* method. After starting the generator, you can send a value to it, which becomes the result of the yield expression within the generator function.

```
def greet():
name = ''
while True:
name = yield "Hello, " + name
# Create a generator object
greeter = greet()
# Initialize the generator
next(greeter)
# Send values into the generator
print(greeter.send('Alice')) # Hello, Alice
print(greeter.send('Bob')) # Hello, Bob
```

➔ In this example, after each *yield*, we can send a new value that will be assigned to the name variable. This technique can be powerful in creating coroutines, which are generalizations of generators and can be used to build asynchronous programs.

➔ An essential practice while working with generators is making sure they clean up resources when they are no longer required. This can be achieved by using generators in conjunction with context managers or by handling the *GeneratorExit* exception, which is raised when a generator's *close()* method is called.

```
def file_reader(file_path):
try:
with open(file_path) as file:
for line in file:
yield line.strip()
except GeneratorExit:
print("Closed the file reader")
fr = file_reader("sample.txt")
# Read a few lines from the file
print(next(fr))
print(next(fr))
# Close the generator, thus releasing any resources it was using
fr.close()
```

➔ In this scenario, we use a generator to read lines from a file one by one. If we decide to stop reading lines before reaching the end of the file, calling *fr.close()* not only terminates the generator but also ensures that the file resource is released properly.

➔ Iterating and consuming generator objects

➔ When we talk about **iterating** and **consuming** generator objects in Python, we are referring to the process of retrieving each value yielded by the generator, until there are no values left to yield. This is typically done using a loop, and the most common loop to use with a generator is the *for* loop.

```
gen = (x * 2 for x in range(5))
for value in gen:
print(value)
# Output: 0, 2, 4, 6, 8
```

➔ This example shows a simple generator expression that doubles each number in a range. As we iterate over the generator object called *gen*, it lazily evaluates and yields the next value in the sequence, until all values have been produced.

➔ Note that once a generator has been consumed (i.e., all its values have been retrieved), it cannot be reset or reused. If you try iterating over it again, it will not yield any more values:

```
for value in gen:
print(value)
# Nothing gets printed as the generator is already exhausted
```

➔ In some cases, you may want to retrieve the next value of a generator without using a *for* loop. This can be done with the next() function, which manually retrieves the next value from a generator:

```
gen = (x * 2 for x in range(5))
print(next(gen)) # Output: 0
print(next(gen)) # Output: 2
```

```
# and so on...
```

➔ Calling next() on a generator will raise a StopIteration exception when no values are left to yield. To handle this gracefully, you can catch this exception in your code:

```
try:
while True:
print(next(gen))
except StopIteration:
pass
# Properly handles StopIteration after printing 0, 2, 4, 6, 8
```

➔ In addition to standard iteration techniques, Python generators can also communicate back and forth with the calling code using the send() method. This advanced feature enables generators to not only produce values but also consume values from outside its scope.

➔ Advanced usage and benefits of Python generators

➔ One of the advanced uses of Python generators is in the implementation of coroutines. Coroutines are a form of concurrency that can be much lighter weight than threading or multiprocessing. Generators enable this by allowing functions to pause and resume their execution at certain points, which can be used to handle I/O-bound jobs efficiently. By using the *asyncio* module, you can write asynchronous code that looks and behaves like synchronous code, but is actually non-blocking. Here's an example:

```
import asyncio
async def fetch_data():
await asyncio.sleep(2)
return {'data': 1}
async def print_numbers():
for i in range(10):
print(i)
await asyncio.sleep(0.25)
async def main():
# Schedule both the coroutines to run
task1 = asyncio.create_task(fetch_data())
task2 = asyncio.create_task(print_numbers())
# Wait for the completion of fetch_data and then print the returned value
value = await task1
print(value)
# Run the main coroutine
asyncio.run(main())
```

➔ Another benefit of generators is their ability to chain together to form pipelines. This can be particularly useful when you need to apply multiple transformations to a stream of data. Ponder this example where we filter out negative values from a dataset and then compute their square roots:

```
import math
# Generator that filters negatives
def no_negatives(data):
for x in data:
if x >= 0:
yield x
# Generator that calculates square roots
```

```
def sqrt(data):
for x in data:
yield math.sqrt(x)
# Data stream as a list
stream = [4, -2, 9, -3, 16, 0]
# Chain the generators
pipeline = sqrt(no_negatives(stream))
# Consume the generator pipeline
for value in pipeline:
print(value)
```

➔ In the snippet above, we do not need to store intermediate results; the data flows through the pipeline as it is being consumed at the end.

➔ Generators also play well with other Python features such as generator expressions and the itertools module, which provide tools for effective looping. With itertools, you can combine generators in very expressive ways using functions like *chain*, *islice*, *cycle*, and more.

```
from itertools import count, islice
# islice can limit an infinite generator
limit_gen = islice(count(), 5) # This will yield 0, 1, 2, 3, 4
for i in limit_gen:
print(i)
```

➔ The **on-demand nature** of generators provides many benefits such as reducing memory footprint, allowing the handling of infinite data streams, and increasing performance in I/O-bound applications or those involving large computational lists. They form a powerful part of the Python language that enables more efficient programming patterns.

➔ In conclusion, advanced usage of Python generators includes implementing coroutines, chaining generators to form pipelines and combining them with other features like generator expressions and itertools module. These usages yield valuable patterns for efficient data processing that are both memory and performance-optimized. As evidenced by these examples, Python's generators are a robust and versatile feature that can open doors to a wealth of powerful programming paradigms.

Que. 2 Difference between yield and return.

➔ Difference between Python yield and Return

S.NO.	YIELD	RETURN
1	Yield is generally used to convert a regular Python function into a generator.	Return is generally used for the end of the execution and "returns" the result to the caller statement.
2	It replace the return of a function to suspend its execution without destroying local variables.	It exits from a function and handing back a value to its caller.

S.NO.	YIELD	RETURN
3	It is used when the generator returns an intermediate result to the caller.	It is used when a function is ready to send a value.
4	Code written after yield statement execute in next function call.	while, code written after return statement wont execute.
5	It can run multiple times.	It only runs single time.
6	Yield statement function is executed from the last state from where the function get paused.	Every function calls run the function from the start.

Que. 3 Understanding iterators and creating custom iterators.

➔ What Is an Iterator?

An iterator is an object that allows you to traverse through all the elements of a collection, such as a list or a tuple, one by one, without needing to know the underlying structure.

Key Components of an Iterator:

__iter__(): This method returns the iterator object itself. It's called when the iteration is initialized, typically by calling iter() on the object.

__next__(): This method returns the next item in the sequence. When there are no more items, it raises the StopIteration exception, signaling that the iteration is complete.

➔ Creating Custom Iterators

You can create custom iterators by defining a class that implements both __iter__() and __next__() methods.

Steps to Create a Custom Iterator:

Define the __iter__() method: This method should return the iterator object itself.

Define the __next__() method: This method should return the next item and raise StopIteration when there are no more items.

7. Functions and Methods

Que. 1 Defining and calling functions in Python.

➔ In Python, defining and calling functions is simple and may greatly improve the readability and reusability of our code. In this article, we will explore How we can define and call a function.

Defining a Function

By using the word def keyword followed by the function's name and parentheses () we can define a function. If the function takes any arguments, they are included inside the parentheses. The code inside a function must be indented after the colon to indicate it belongs to that function.

Syntax of defining a function:

def function_name(parameters):

 # Code to be executed

return value

- **function_name:** The name of your function.
- **parameters:** Optional. A list of parameters (input values) for the function.
- **return:** Optional. The return statement is used to send a result back from the function.
- **fun():** A simple function that prints "Welcome to GFG" when called, without any parameters.
- **greet(name, age):** A function with two parameters (name and age) that prints the values passed to it when called.

- ## Calling a Function

- To call a function in Python, we definitely type the name of the function observed via parentheses (). If the function takes any arguments, they may be covered within the parentheses . Below is the example for calling def function Python.
- **Syntax of Calling a function:**
- *function_name(arguments)*
- **Calling fun()**: The function fun() is called without any arguments, and it prints "Welcome to GFG".
- **Calling greet("Alice", 21)**: The function greet() is called with the arguments "Alice" and 21, which are printed as "Alice 21".

Que. 2 Function arguments (positional, keyword, default).

➔ In this tutorial we will learn different ways to pass arguments to a function.

There are different ways to pass arguments to a Python function. They are as follows:

- Positional Arguments
- Default Arguments
- Keyword Arguments

Python Positional Arguments

The positional arguments are the most basic type of arguments passed to a function.

When you call a function and provide values for its parameters, those values are assigned to the parameters based on their position or order in the function's parameter list. For example,

We have called the add_numbers() function with two arguments.

- **5.4** - assigned to the first parameter n1
- **6.7** - assigned to the second parameter n2

These arguments are passed based on their position. Hence, are called positional arguments.

If we remove the second argument, it will lead to an error message:

The code throws the following error:

This error message tells us that we must pass a second argument for n2 during the function call because our add_numbers() function takes **2** arguments.

Python Default Arguments

Function arguments can have default values in Python. The default arguments are parameters in a function definition that have a predefined value assigned to them.

These default values are used when an argument for that parameter is not provided during the function call.

We can provide a default value to an argument by using the assignment operator =. For example,

Here, n1 takes the value passed in the function call. Since there is no second argument in the function call, n2 takes the default value of **1000**.

We can also run the function without any arguments:

Keyword Arguments

In Python, keyword arguments are a way to pass arguments to a function using the parameter names explicitly.

Instead of relying on the order of the arguments, keyword arguments allow you to specify the argument along with its corresponding parameter name.

During the second function call greet(message='Howdy?', name='Jill'), we passed arguments by their parameter name.

The string 'Howdy?' is associated with the message parameter, and the string 'Jill' is associated with the name parameter. When we call functions in this way, the order (position) of the arguments can be changed.

Que. 3 Scope of variables in Python.

➔ Local Scope
Variables declared inside a function are local to that function.
They can only be accessed within the function where they are defined.

```
def greet():
    message = "Hello, World!"  # Local variable
    print(message)

greet()
# print(message)  # NameError: name 'message' is not defined
```

➔ Global Scope
Variables declared outside of any function or block have a global scope.
They can be accessed from any part of the program.

```
greeting = "Hello"  # Global variable

def greet():
    print(greeting)  # Accessing the global variable

greet()
print(greeting)  # Global variable can be accessed outside the function
```

➔ Enclosing Scope (Nonlocal Scope)

Enclosing scope applies to nested functions.
Variables in an outer (enclosing) function are accessible to inner functions.
To modify them in the inner function, use the nonlocal keyword.

```
def outer():
    message = "Outer scope"

    def inner():
        nonlocal message
        message = "Modified in inner scope"
        print("Inner:", message)

    inner()
    print("Outer:", message)

outer()
```

Que. 4 Built-in methods for strings, lists, etc.

➔ String Methods

str.lower(): Converts to lowercase.
str.upper(): Converts to uppercase.
str.capitalize(): Capitalizes the first letter.
str.title(): Capitalizes the first letter of each word.
str.strip(): Removes leading and trailing spaces (or specified characters).
str.replace(old, new): Replaces all occurrences of old with new.

str.startswith(sub): Returns True if the string starts with sub.
str.endswith(sub): Returns True if the string ends with sub.
str.isalpha(): Returns True if all characters are alphabetic.
str.isdigit(): Returns True if all characters are digits.
str.isspace(): Returns True if all characters are whitespace.

➔ List Methods

list.append(x): Adds an item to the end.
list.insert(i, x): Inserts an item at position i.
list.pop([i]): Removes and returns the item at position i (last item if not specified).
list.remove(x): Removes the first occurrence of x.

list.clear(): Removes all elements.

list.index(x): Returns the index of the first occurrence of x.
list.count(x): Counts occurrences of x.
len(lst): Returns the number of elements in the list.

list.sort(): Sorts the list in ascending order (in-place).
list.reverse(): Reverses the order of the list (in-place).
sorted(lst): Returns a new sorted list.

8. Control Statements (Break, Continue, Pass)

Que 1. Understanding the role of break, continue, and pass in Python loops.

➔ break Statement

The break statement is used to exit a loop prematurely.
When break is executed, the loop stops immediately, and control moves to the statement following the loop.

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

➔ continue Statement

The continue statement is used to skip the rest of the code in the current iteration of the loop and proceed to the next iteration.

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

➔ pass Statement

The pass statement is a placeholder.
It does nothing and allows the loop or function to run without errors when no action is specified.

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

```
        pass  # Placeholder for future logic
    print(i)
```

9. String Manipulation

Que 1. Understanding how to access and manipulate strings.

➔ Accessing Strings

➔ Indexing

Strings can be accessed using index numbers.
The index starts from 0 for the first character and goes up to n-1 for the last character.
Negative indices can be used to access characters from the end.

```
s = "Python"
print(s[0])  # 'P' (First character)
print(s[-1]) # 'n' (Last character)
```

➔ Slicing

Slicing allows you to extract a substring using the syntax:
string[start:end:step]

- start: Index to begin slicing (inclusive).
- end: Index to stop slicing (exclusive).
- step: Interval between characters.

```
s = "Python"
print(s[1:4])   # 'yth' (Characters at indices 1, 2, 3)
print(s[:3])    # 'Pyt' (First 3 characters)
print(s[::2])   # 'Pto' (Every 2nd character)
print(s[::-1])  # 'nohtyP' (Reversed string)
```

➔ Manipulating Strings

➔ Concatenation

Combine two or more strings using the + operator.

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)  # 'Hello World'
```

➔ Repetition

Repeat a string multiple times using the * operator.

```
s = "Python"
print(s * 3)  # 'PythonPythonPython'
```

Que 2.Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

➔ Concatenation

Concatenation is the process of combining two or more strings using the + operator.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: "Hello World"
```

➔ Repetition

Repetition involves repeating a string multiple times using the * operator.

```
str1 = "Hi! "
result = str1 * 3
print(result)  # Output: "Hi! Hi! Hi! "
```

➔ String Methods

str.lower(): Converts to lowercase.
str.upper(): Converts to uppercase.
str.capitalize(): Capitalizes the first letter.
str.title(): Capitalizes the first letter of each word.
str.strip(): Removes leading and trailing spaces (or specified characters).
str.replace(old, new): Replaces all occurrences of old with new.
str.startswith(sub): Returns True if the string starts with sub.
str.endswith(sub): Returns True if the string ends with sub.
str.isalpha(): Returns True if all characters are alphabetic.
str.isdigit(): Returns True if all characters are digits.
str.isspace(): Returns True if all characters are whitespace.

Que 3. String slicing

➔ String slicing in Python allows you to extract parts of a string (substrings) using a specific syntax:
string[start:end:step]

start: The index where the slice starts (inclusive). Default is 0.
end: The index where the slice ends (exclusive). Default is the length of the string.
step: The interval between each character in the slice. Default is 1.

➔ Simple Slicing

```
s = "Python"

print(s[0:3])  # 'Pyt' (characters at indices 0, 1, 2)
print(s[2:5])  # 'tho' (characters at indices 2, 3, 4)
print(s[:4])   # 'Pyth' (start defaults to 0)
print(s[3:])   # 'hon' (end defaults to length of the string)
print(s[:])    # 'Python' (full string)
```

➔ Negative Indices

```
s = "Python"

print(s[-4:-1])  # 'tho' (characters from -4 to -2)
print(s[-3:])    # 'hon' (last three characters)
print(s[:-3])    # 'Pyt' (all except the last three characters)
```

➔ Skipping Characters (Step)

```
s = "Python"

print(s[::2])   # 'Pto' (every second character)
print(s[1::2])  # 'yhn' (every second character starting from index 1)
print(s[::-1])  # 'nohtyP' (reversed string)
print(s[::-2])  # 'nhy' (every second character in reverse order)
```

➔ Slicing Patterns

1. Extract Substring

```
s = "Programming"
substring = s[0:6]
print(substring)  # 'Progra'
```

2. Reverse a String

```
s = "Python"
reversed_s = s[::-1]
```

```
print(reversed_s)  # 'nohtyP'
```

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

Que 1.How functional programming works in Python.

➔ Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data.

➔ Pure Functions

Functions that always produce the same output for the same input and have no side effects.

They don't modify external variables or depend on external state.

```
# Pure function
def add(a, b):
    return a + b

# Impure function (depends on external variable)
x = 10
def add_impure(a):
    return a + x
```

➔ First-Class Functions

Functions in Python are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

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

# Assign to a variable
func = square
print(func(5))  # 25

# Pass as an argument
def apply_function(f, value):
    return f(value)
```

```
print(apply_function(square, 6))  # 36
```

➔ Immutability

Functional programming emphasizes immutability, where data is not modified but rather new data is created.

Python does not enforce immutability but supports immutable data types like tuples and strings.

```
# Instead of modifying a list:
nums = [1, 2, 3]
new_nums = [x + 1 for x in nums]  # Creates a new list
print(new_nums)  # [2, 3, 4]
```

➔ Lambda Functions

Anonymous, single-expression functions.

Commonly used in map(), filter(), and reduce().

```
# Traditional function
def add(x, y):
    return x + y

# Lambda function
add_lambda = lambda x, y: x + y
print(add_lambda(2, 3))  # 5
```

Que 2. Using map(), reduce(), and filter() functions for processing data.

➔ map()

Applies a function to each element in an iterable and returns a map object (which can be converted to a list, tuple, etc.).

Syntax:

```
map(function, iterable)
```

➔ filter()

Filters elements from an iterable based on a condition defined in a function.

It returns a filter object containing elements where the condition is True.

Syntax:

filter(function, iterable)

➔ reduce()

Aggregates elements in an iterable into a single value using a function.
Unlike map() and filter(), reduce() is not a built-in function and must be imported from the functools module.

Syntax:

from functools import reduce
reduce(function, iterable, initializer=None)

Que 3. Introduction to closures and decorators.

➔ Closures

A closure is a function that remembers its lexical scope, even when the function is called outside that scope.
In simpler terms, closures occur when a nested function references variables from its enclosing function, and the outer function has finished execution.
The inner function "remembers" the environment in which it was created, including the values of the variables.

➔ Decorators

A decorator is a function that takes another function as an argument, extends or alters its behavior, and then returns a new function.
Decorators allow you to modify or enhance the behavior of functions or methods without changing their actual code.

Decorators are commonly used for:

Logging function calls
Timing functions
Adding access control or authentication

How Decorators Work:

A decorator is defined as a function that accepts a function as an argument.
It defines a new function that wraps (or enhances) the original function.
The new function is returned, and when the decorated function is called, the new behavior is applied.