Q1. What is the difference between a function and a method in Python?

**Function**:- A function is a block of code that is defined using the def keyword. It can be called directly by its name and is not inherently tied to any object.

Functions can exist independently of any object or class.

Example: def my\_function(x): return x + 1

**Method:-** A method is a function that is associated with an object (an instance of a class) or a class itself. It is defined within a class and is called on an instance of that class or the class itself.

Methods have access to the instance or class they are associated with through the self (for instance methods) or cls (for class methods) parameters.

Example: class MyClass:

def my method(self, x):

return x + 1

Q2. Explain the concept of function arguments and parameters in Python.

**Parameters :-** Parameters are variables listed in the function definition. They act as placeholders for the values that will be passed into the function when it is called.Parameters are defined within the parentheses of the function definition.

def greet(name, age):

print(f"Hello {name}, you are {age} years old!")

**Arguments** :- Arguments are the actual values or expressions passed to a function when it is called.

They provide the real data that the function uses, replacing the parameters.

greet("Rohan", 30)

Q3. What are the different ways to define and call a function in Python?

Use the def keyword to define a function.

Call: Call the function by using its name followed by parentheses.

Example: def greet(name):

print(f"Hello, {name}!")

greet("Ajay")

Default Parameter :- Parameters can have default values. If no argument is provided for these parameters, the default values are used.

Call: You can omit the default parameters when calling the function.

Example: def greet(name, age=25):

print(f"Hello, {name}. You are {age} years old.")

greet("Manisha")

greet("Rahul", 30)

Function with Variable-Length Arguments:- Use \*args to accept a variable number of positional arguments, and \*\*kwargs to accept a variable number of keyword arguments.

Call: Pass multiple positional or keyword arguments.

Example: def print\_args(\*args, \*\*kwargs):

print("Positional arguments:", args)

print("Keyword arguments:", kwargs)

print\_args(1, 2, 3, a=4, b=5)

Lambda Functions (Anonymous Functions) :- Use the lambda keyword to create small, anonymous functions. Lambda functions are used for short operations where defining a full function might be overkill.

Call: Call the lambda function just like a regular function.

Example: add = lambda x, y: x + y

print(add(3, 5))

Nested Functions:- Functions can be defined inside other functions. The inner function is local to the outer function and cannot be accessed from outside.

Call: Call the inner function from within the outer function.

Example: def outer\_function(x):

def inner\_function(y):

return y \* y

return inner\_function(x) + 1

print(outer\_function(3))

Functions as First-Class Objects :- In Python, functions are first-class objects, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

Call: Invoke functions just like any other object.

Example: def square(x):

return x \* x

def apply\_function(func, value):

return func(value)

print(apply\_function(square, 5))

Recursive Functions :- A function can call itself, which is useful for solving problems that can be broken down into simpler, similar problems.

Call: The function calls itself within its own body.

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5))

Function Annotations:- Annotations provide a way to attach metadata to function arguments and return values. They are optional and do not affect the function’s behavior.

Call: Annotations are accessed via the \_\_annotations\_\_ attribute of the function.

Example: def greet(name: str, age: int) -> str:

return f"Hello, {name}. You are {age} years old."

print(greet("chunky", 30))

print(greet.\_\_annotations\_\_)

Q4. What is the purpose of the `return` statement in a Python function?

Purpose of the return Statement

Returning a Value:- The primary purpose of the return statement is to send a result back to the caller of the function. This allows the function to produce an output that can be used by the rest of the program.

Exiting a Function:- The return statement terminates the execution of the function. When the return statement is encountered, the function ends immediately, and control is returned to the point where the function was called.

Returning Multiple Values:- Functions can return multiple values as a tuple. Python's ability to return multiple values makes it convenient to package related data together.

Returning None:- If a function does not have a return statement or the return statement does not include an expression, the function returns None by default. This indicates the absence of a meaningful return value.

Examples :Basic Return:

def add(a, b):

return a + b

result = add(3, 5)

print(result) # Output: 8

In this example, the return statement sends the sum of a and b back to the caller. The function call add(3, 5) evaluates to 8.

Returning Multiple Values:

def get\_person\_info():

name = "Alice"

age = 30

return name, age

person = get\_person\_info()

print(person) # Output: ('Alice', 30)

In this example, the function returns a tuple containing both name and age.

Early Exit with Return:

def find\_max(num\_list):

if not num\_list:

return None # Early exit if the list is empty

max\_num = num\_list[0]

for num in num\_list:

if num > max\_num:

max\_num = num

return max\_num

result = find\_max([1, 5, 2, 8, 3])

print(result) # Output: 8

Here, the return statement is used to exit the function early if the input list is empty. Otherwise, it returns the maximum value from the list.

Implicit Return of None:

def print\_message(message):

print(message)

Call: result = print\_message("Hello, World!")

print(result) # Output: None

Since the function print\_message does not have a return statement, it implicitly returns None. This is evident when result is printed.

Q5. What are iterators in Python and how do they differ from iterables?

In Python, iterators and iterables are fundamental concepts used for traversing and processing sequences of data. Although they are closely related, they serve different purposes and have distinct characteristics.

**Iterables** :- An iterable is any Python object capable of returning its members one at a time. You can use an iterable in a for loop or any context where an iterable is expected.

Key Characteristics:

An Iterable Must Implement the \_\_iter\_\_ Method:

This method returns an iterator object.

Examples of Iterables:

Lists, tuples, dictionaries, sets, and strings are common examples of iterables. They all implement the \_\_iter\_\_ method, which allows you to iterate over their elements.

Example:

A list is an iterable

my\_list = [1, 2, 3, 4]

for item in my\_list:

print(item)

In this example, my\_list is an iterable because it supports iteration.

**Iterators**:- An iterator is an object that represents a stream of data. It is used to iterate through the elements of an iterable. Iterators are created from iterables and provide a way to access the elements of the iterable one at a time.

Key Characteristics:

An Iterator Must Implement Two Methods:

\_\_iter\_\_(): This method returns the iterator object itself. It is used to initialize the iterator.

\_\_next\_\_(): This method returns the next item from the stream. When there are no more items, it raises the StopIteration exception to signal the end of iteration.

Iterators Do Not Need to Support Indexing:Unlike lists or tuples, iterators do not support indexing. They are designed to be used sequentially.

Example:

Create an iterator from a list

my\_list = [1, 2, 3, 4]

my\_iterator = iter(my\_list)

Iterate using the iterator

print(next(my\_iterator)) OPT- 1

print(next(my\_iterator)) OPT -2

Q6. Explain the concept of generators in Python and how they are defined.

Concept of Generator

Lazy Evaluation:

Generators produce items one at a time and only when requested, rather than computing all items upfront. This is known as lazy evaluation, which is particularly useful for working with large datasets or streams of data.

State Preservation:

Generators automatically maintain their state between iterations. This means that each time a generator’s yield statement is executed, it pauses and saves its state, allowing it to resume where it left off.

Memory Efficiency:

Because they generate items on the fly, generators are more memory-efficient than lists or other data structures that require storing all items in memory at once.

Single Iteration:

Generators can only be iterated once. After all items have been generated, the generator cannot be reused. If you need to iterate again, you must create a new generator.

Defining a Generator

Generators are defined using functions that contain one or more yield statements. When the function is called, it returns a generator object without executing the function body immediately. The yield statement produces a value and pauses the function’s execution, allowing it to be resumed later from the same point.

def generator\_function():

yield value1

yield value2

# More yield statements

Example:

Here’s a simple example of a generator that yields numbers from 1 to 3:

def number\_generator():

yield 1

yield 2

yield 3

Q7. What are the advantages of using generators over regular functions?

Generators in Python offer several advantages over regular functions, particularly when it comes to handling large datasets or streams of data. Here are the key advantages of using generators:

Memory Efficiency

Advantage: Generators are more memory-efficient compared to regular functions that return complete collections (like lists). This is because generators produce items one at a time and only when needed, rather than storing all items in memory at once.

Example: def generate\_large\_sequence():

for i in range(10\*\*6):

yield i

Using the generator

gen = generate\_large\_sequence()

for num in gen:

Process the number

pass

In this example, generate\_large\_sequence does not create a large list in memory; instead, it generates each number one by one.

Lazy Evaluation

Advantage: Generators use lazy evaluation, meaning they compute values only when requested. This can improve performance and reduce the time to start processing since you do not need to wait for all values to be generated upfront.

Example def count\_up\_to(max):

count = 1

while count <= max:

yield count

count += 1

Process only the first few numbers

gen = count\_up\_to(1000000)

for i in range(5):

print(next(gen)) # Only computes and prints first 5 numbers

Here, values are produced only as they are requested, not all at once.

Simplified Code for Complex Iterations

Advantage: Generators can simplify code, especially when dealing with complex iteration patterns or streams of data. They allow you to write concise and readable code without manually managing the state of iteration.

Example:

def fibonacci(n):

a, b = 0, 1

for \_ in range(n):

yield a

a, b = b, a + b

Using the generator

for number in fibonacci(10):

print(number)

In this example, the generator handles the complexity of the Fibonacci sequence iteration internally.

Infinite Sequences

Advantage: Generators can represent infinite sequences. Since they yield values one at a time, they can produce an infinite series without running out of memory or requiring an infinite amount of storage.

Example:

def infinite\_numbers():

n = 1

while True:

yield n

n += 1

Using the generator

gen = infinite\_numbers()

for \_ in range(5):

print(next(gen)) # Prints the first 5 numbers

In this case, the generator produces numbers indefinitely without consuming large amounts of memory.

Better Performance for Large Data

Advantage: Generators can improve performance when processing large datasets by avoiding the overhead of creating and storing large lists or collections.

Example:

def large\_file\_lines(file\_path):

with open(file\_path) as file:

for line in file:

yield line.strip()

Processing lines one at a time

for line in large\_file\_lines('large\_file.txt'):

Process each line

pass

Here, each line is read and processed one by one without loading the entire file into memory.

State Preservation

Advantage: Generators maintain their state between iterations. This allows them to remember where they left off and resume from that point, which is useful for managing sequences or streams of data.

Example:

def reverse\_words(text):

words = text.split()

while words:

yield words.pop()

Using the generator

gen = reverse\_words("hello world")

print(next(gen)) # Output: world

print(next(gen)) # Output: hello

Q8. What is a lambda function in Python and when is it typically used?

A lambda function in Python is a small, anonymous function defined with the lambda keyword. Unlike regular functions defined with def, lambda functions are used for short-lived, simple operations where a full function definition might be overkill.

Lambda Function Syntax

The syntax for a lambda function is:

lambda arguments: expression

lambda: The keyword used to define the lambda function.

arguments: A comma-separated list of input parameters.

expression: A single expression that is evaluated and returned.

Example:

Lambda function that adds 10 to its input

add\_ten = lambda x: x + 10

Using the lambda function

print(add\_ten(5)) # Output: 15

In this example, lambda x: x + 10 creates a lambda function that takes one argument x and returns x + 1

Typical Use Cases for Lambda Functions

Short-Term Use:

Lambda functions are often used when you need a simple function for a short period, especially when defining a full function is unnecessarily verbose.

Example:

Using lambda to create a small function on the fly

print((lambda x, y: x \* y)(3, 4)) # Output: 12

As Arguments to Higher-Order Functions:

Lambda functions are commonly used as arguments to higher-order functions such as map(), filter(), and sorted(), where you need to apply a function to each item in a collection or specify a custom sort order.

Example with map():

numbers = [1, 2, 3, 4]

doubled = map(lambda x: x \* 2, numbers)

print(list(doubled)) # Output: [2, 4, 6, 8]

Example with filter():

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even\_numbers)) # Output: [2, 4, 6]

Example with sorted():

points = [(1, 2), (3, 1), (5, 4)]

sorted\_points = sorted(points, key=lambda p: p[1])

print(sorted\_points) # Output: [(3, 1), (1, 2), (5, 4)]

Inline Function Definitions:

Lambda functions are useful when you need to define a small function quickly within another function or context, avoiding the need to formally define a new function with def.

Example:

def apply\_function(func, value):

return func(value)

result = apply\_function(lambda x: x \*\* 2, 4)

print(result) # Output: 16

Creating Small Callbacks:

Lambda functions are often used to create small, anonymous callback functions for event handling, sorting, or other tasks where a short, one-off function is needed.

Example in a GUI library:

button = Button(text="Click me")

button.bind("<Button-1>", lambda e: print("Button clicked!"))

Q9. Explain the purpose and usage of the `map()` function in Python

The map() function in Python is a built-in higher-order function that allows you to apply a function to each item in an iterable (like a list, tuple, or set) and return an iterator that produces the results. It’s a powerful tool for transforming data in a concise and efficient way.

Purpose of map()

Transform Data: map() is used to apply a function to each item in an iterable, transforming the data according to the function’s logic.

Simplify Code: It provides a concise way to perform operations on collections without the need for explicit loops.

Syntax of map()

map(function, iterable, ...)

function: A function that takes one or more arguments and returns a value. This function is applied to each item in the iterable.

iterable: An iterable (e.g., a list, tuple, or set) whose elements are to be processed by the function.

You can pass multiple iterables to map(). In this case, the function must accept as many arguments as there are iterables.

Basic Example

Here’s a basic example of using map() to apply a function to each element in a list:

Example:

def square(x):

return x \* x

numbers = [1, 2, 3, 4]

squared\_numbers = map(square, numbers)

Convert the map object to a list to view the results

print(list(squared\_numbers)) # Output: [1, 4, 9, 16]

In this example:

The square function is applied to each number in the numbers list.

map() returns an iterator, which we convert to a list to see the results.

Using Lambda Functions with map()

Lambda functions are often used with map() for short, simple transformations without the need to define a separate function.

Example:numbers = [1, 2, 3, 4]

squared\_numbers = map(lambda x: x \* x, numbers)

print(list(squared\_numbers)) # Output: [1, 4, 9, 16]

Here, lambda x: x \* x serves as a concise way to square each number in the numbers list.

Multiple Iterables

When passing multiple iterables to map(), the function should take as many arguments as there are iterables. map() applies the function to items from all iterables in parallel, stopping when the shortest iterable is exhausted.

Example:

def add(x, y):

return x + y

list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = map(add, list1, list2)

print(list(result)) # Output: [5, 7, 9]

In this example:

The add function is applied to pairs of elements from list1 and list2.

Handling Edge Cases

Empty Iterable: If the iterable is empty, map() returns an empty iterator.

Example:

def identity(x):

return x

empty\_list = []

result = map(identity, empty\_list)

print(list(result)) # Output: []

Function Argument Mismatch: If the function does not match the number of iterables provided, a TypeError will be raised.

Example:

def add(x, y, z):

return x + y + z

list1 = [1, 2]

list2 = [3, 4]

try:

result = map(add, list1, list2)

print(list(result))

except TypeError as e:

print(e) OPT add() missing 1 required positional argument: 'z'

Q10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

The map(), reduce(), and filter() functions in Python are all higher-order functions used for processing and transforming iterables. Although they share some similarities, they serve distinct purposes and operate in different ways. Here’s a detailed comparison of these functions:

map() Function

Purpose: The map() function applies a given function to each item of an iterable (e.g., list, tuple) and returns an iterator that produces the results.

map(function, iterable, ...)

Characteristics: Transformative: It transforms each element of the iterable using the provided function.

Multiple Iterables: It can accept multiple iterables, applying the function in parallel to corresponding items.

Returns Iterator: It returns an iterator, which can be converted to a list or other collections to view the results.

Example: numbers = [1, 2, 3, 4]

squared = map(lambda x: x \*\* 2, numbers)

print(list(squared)) # Output: [1, 4, 9, 16]

reduce() Function

Purpose: The reduce() function applies a binary function (a function that takes two arguments) cumulatively to the items of an iterable, from left to right, to reduce the iterable to a single value.

from functools import reduce

reduce(function, iterable, [initializer])

Characteristics: Reduction: It reduces the iterable to a single accumulated result based on the binary function.

Optional Initializer: An optional initializer can be provided to start the accumulation.

Not Built-In: In Python 3, reduce() is not a built-in function; it must be imported from the functools module.

Example:

from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 24

filter() Function

Purpose: The filter() function filters items from an iterable based on a provided function that returns either True or False, returning an iterator of the items that evaluate to True.

filter(function, iterable)

Characteristics: Filtering: It filters elements from the iterable based on a boolean condition provided by the function.

Returns Iterator: It returns an iterator containing only the elements for which the function returns True.

Example: numbers = [1, 2, 3, 4, 5]

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even\_numbers)) # Output: [2, 4]

Q11. Using pen & Paper write the internal mechanism for sum operation using  reduce function on this given list:[47,11,42,13];

