Start coding or generate with AI.

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

   In Python, functions and methods are both callable objects that perform actions, but they have key differences in their usage and context:

2. Function A function is a block of code that performs a specific task and is defined using the def keyword. It is independent and can be called directly by its name. Functions can take zero or more arguments and return a value. Example:

```
def greet(name):
    return f"Hello, {name}!"

result = greet("Sakshi")


print(result)
```

```
Hello, Sakshi!
```

Method A method is a function that is associated with an object (typically an instance of a class) and operates on that object. Methods are called using the syntax object.method(), where object is an instance of a class, and method is the function defined within that class. Methods always take at least one argument, which is the object itself (typically referred to as self), and they can perform actions on the object's data.

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

    def greet(self):
        return f"Hello, {self.name}!"


g = Greeter("Sakshi")
result = g.greet()  # Calling the method
print(result)
```

```
Hello, Sakshi!
```

Key Differences: Context:

A function is independent and not tied to any object. A method is a function defined within a class and operates on instances of that class. Calling:

A function is called directly by its name: function(). A method is called on an object: object.method(). Implicit argument:

A function does not require an implicit argument (it can take explicit arguments). A method always takes self (or cls for class methods) as the first argument, which refers to the instance or class the method is associated with.

2. Explain the concept of function arguments and parameters in Python?

In Python, function parameters and arguments are related to how data is passed to functions:

Parameters are the variables defined in a function's definition. They act as placeholders for the values that will be passed to the function.

```
def greet(name):  # 'name' is the parameter
    return f"Hello, {name}!"


message = greet("Sakshi")
print(message)
```

```
Hello, Sakshi!
```

Arguments are the actual values passed to the function when it is called. These values are assigned to the corresponding parameters.

```
result = greet("Sakshi")  # "Alice" is the argument


print(result)
```

```
Hello, Sakshi!
```

Types of Arguments: Positional Arguments: Passed in a specific order.

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

```
add(55,66)
```

```
121
```

Keyword Arguments: Passed by specifying the parameter names.

```
def describe_pet(animal_type, pet_name):
    return f"I have a {animal_type} named {pet_name}."
```

```
result = describe_pet(animal_type="dog", pet_name="Buddy")
print(result)
```

```
I have a dog named Buddy.
```

Default Arguments: Parameters can have default values, which are used if no argument is passed.

```
def greet(name, message="Hello"):
    return f"{message}, {name}!"
```

```
result1 = greet("Sakshi", "Good morning")
print(result1)
```

```
Good morning, Sakshi!
```

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

   Basic Function Define: Use def with parameters. Call: Call by name with arguments.

```
def greet(name): return f"Hello, {name}!"
greet("ABC")
```

```
'Hello, ABC!'
```

Default Arguments Define: Set default values for parameters. Call: Omit arguments for default parameters.

```
def greet(name, message="Hello"):
   return f"{message}, {name}!"
greet("ABC")
```

```
'Hello, ABC!'
```

Variable-Length Arguments Define: Use *args or **kwargs. Call: Pass any number of arguments.

```
def print_numbers(*args):
   print(*args)
print_numbers(1, 2, 3)
```

```
1 2 3
```

Lambda (Anonymous) Function Define: Use lambda for a small function. Call: Directly call the lambda.

```
multiply = lambda x, y: x * y
multiply(3, 4)
```

```
12
```

Function with Return Values, Multiple Return Values,Function as Argument

```
multiply = lambda x, y: x * y
multiply(3, 4)
```

```
12
```

```
def add(a, b): return a + b
add(5, 3)
```

```
8
```

```
def get_dimensions(): return 10, 20
width, height = get_dimensions()
```

```
def apply_function(f, x): return f(x)
apply_function(lambda n: n * n, 5)
```

```
25
```

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

The return statement in a Python function is used to send a result or value back to the caller of the function. It ends the function's execution and specifies what value should be returned.

1. To Send a Value Back to the Caller When a function is called, it can process data and return a result to the part of the program that called it. The return statement specifies the value that will be sent back.

The return statement can provide output from the function to be used elsewhere in the program. For example:

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

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

```
8
```

To End the Function Execution The return statement immediately terminates the function, even if there are other lines of code following it. Any code after a return statement in a function will not be executed.

```
def test_function():
    return "End of function"
    print("This will never be printed")

print(test_function())
```

```
End of function
```

To Return Multiple Values In Python, the return statement can return multiple values as a tuple, which can be unpacked by the caller.

Double-click (or enter) to edit

```
def divide(a, b):
    if b != 0:
        return a // b, a % b  # Returns quotient and remainder
    else:
        return None, None  # Handles division by zero

quotient, remainder = divide(10, 3)
print(quotient, remainder)
```

```
3 1
```

To Provide a Default Value If a function doesn't include a return statement, it returns None by default.

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

result = greet()  # The function prints "Hello!" but returns None
print(result)
```

```
Hello!
None
```

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

   Iterables Definition: An iterable is any object that can be looped over (iterated) using a for loop or other iteration constructs. Examples include lists, tuples, strings, dictionaries, and sets. Key Characteristic: An iterable must implement the **iter**() method, which returns an itertor

```
my_list = [1, 2, 3,4]  # A list is an iterable
for item in my_list:
    print(item)
```

```
1
2
3
4
```

Iterators Definition: An iterator is an object that represents a stream of data. It produces one item at a time when you call the next() function on it. Key Characteristics: An iterator implements two methods: **iter**(): Returns the iterator object itself. **next**(): Returns the next item in the sequence. Raises a StopIteration exception when no more items are available. It is stateful, meaning it "remembers" its position during iteration.

```
my_iterator = iter([1, 2, 3])  # Create an iterator
print(next(my_iterator))  # Output: 1
print(next(my_iterator))  # Output: 2
print(next(my_iterator))
```

```
1
2
3
```

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

Generators in Python A generator in Python is a special type of iterator that is used to produce a sequence of values lazily (on-demand) rather than computing all the values at once and storing them in memory. Generators are particularly useful for working with large datasets or infinite sequences.

Defining Generators Using a Function with yield:

A generator is defined using a function containing one or more yield statements instead of return. The yield keyword produces a value and pauses the generator's state, allowing it to resume from where it left off when called again.

```
Start coding or generate with AI.
```

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()  # Creates a generator object
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

```
1
2
3
```

Using Generator Expressions:

Similar to list comprehensions but produce values lazily.

Double-click (or enter) to edit

```
gen_expr = (x**2 for x in range(25))  # A generator expression
print(next(gen_expr))
print(next(gen_expr))
```

```
0
1
```

Return vs Yield: A regular function uses return, which ends the function and returns a value. A generator uses yield, which pauses the function and allows it to resume.

Advantages of Generators Memory Efficient: Since they generate items one by one, they are ideal for large datasets. Simpler Code: Easier to write compared to manually implementing an iterator. Infinite Sequences: Can model infinite sequences without consuming memory upfront.

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

Memory Efficiency: Generators produce items one at a time, consuming minimal memory compared to creating and storing all items at once in a list or similar data structure.

Lazy Evaluation: Values are generated on-demand, which saves computation time when not all items are needed.

Simpler Code: Generators are easier to implement using the yield keyword compared to manually defining iterator classes with **iter**() and **next**().

Supports Infinite Sequences: Generators can model infinite sequences (e.g., Fibonacci numbers) without exhausting memory.

Pipeline Processing: Generators can be chained to process data in steps, improving modularity and efficiency.

Improved Performance: Avoids the overhead of building large data structures, leading to faster execution for certain tasks.

One-Time Use: Generators are stateful, making them ideal for sequential data processing without retaining unnecessary data.

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

A lambda function in Python is an anonymous, single-expression function defined with the lambda keyword. It is typically used for short, throwaway functions in situations where defining a full function with def is unnecessary.

```
square = lambda x: x**2
print(square(4))
```

```
16
```

Common Uses: With higher-order functions: map(), filter(), sorted(): python Copy code

```
nums = [1, 2, 3, 4]
squares = map(lambda x: x**2, nums)
print(list(squares))
```

```
[1, 4, 9, 16]
```

```
# Custom sorting:
names = ["Alice", "Bob", "Charlie"]
sorted_names = sorted(names, key=lambda name: len(name))
print(sorted_names)
```

```
['Bob', 'Alice', 'Charlie']
```

Advantages: Concise and quick to write. Ideal for short, one-time use. Limitations: Limited to a single expression. Harder to read for complex operations.

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

The map() function in Python is used to apply a specified function to each item in an iterable (like a list, tuple, etc.) and return an iterator (map object) that yields the results. It is often used for transforming or modifying data in a concise way.

function: A function that is applied to each element in the iterable. iterable: An iterable (e.g., list, tuple) whose elements are passed to the function.

Double-click (or enter) to edit

```
nums = [1, 2, 3, 4]
squared = map(lambda x: x**2, nums)
print(list(squared))
```

```
[1, 4, 9, 16]
```

Usage: Transform Data: Apply a function to each element of an iterable. Concise: Avoids the need for writing explicit loops.

Typical Use Cases: Modifying or transforming elements in a list. Applying a function to each item in a sequence (e.g., square each number, convert strings to uppercase). Advantages: More concise and often more efficient than using a loop. Returns an iterator, which is memory-efficient. In short, map() is a powerful function for transforming data by applying a function to each item in an iterable.

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

map():

Purpose: Applies a function to each item in an iterable and returns an iterator with the results. Usage: Transformation of each item. Returns: Transformed values. reduce():

Purpose: Applies a function cumulatively to the items of an iterable, reducing it to a single value. Usage: Accumulation or reduction of values. Returns: A single value. filter():

Purpose: Filters items in an iterable based on a condition (function returns True or False). Usage: Selecting items that satisfy a condition.

```
numbers = [1, 2, 3, 4]
result = map(lambda x: x * 2, numbers)  # Doubles each number
print(list(result))
```

```
[2, 4, 6, 8]
```

reduce(): Purpose: Applies a binary function (a function that takes two arguments) cumulatively to the items of an iterable, reducing it to a single value. Output: Returns a single value, which is the result of applying the binary function to the iterable elements. Usage: When you want to accumulate or reduce the elements in an iterable into a single result (e.g., summing a list of numbers or multiplying all elements).

```
numbers = [1, 2, 3, 4]
from functools import reduce
result = reduce(lambda x, y: x + y, numbers)
print(result)
```

```
10
```

filter(): Purpose: Filters elements from an iterable based on a condition (function). Only the items for which the function returns True are kept. Output: Returns an iterator containing the elements that satisfy the condition. Usage: When you want to filter or exclude elements from an iterable based on a specific condition.

```
numbers = [1, 2, 3, 4, 5, 6]
result = filter(lambda x: x % 2 == 0, numbers)  # Keeps only even numbers
print(list(result))
```

```
[2, 4, 6]
```

Practical Questions:

1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.

```
def sum_of_even_numbers(numbers):
  return sum(num for num in numbers if num % 2 == 0)



numbers = [1, 2, 3, 4, 5, 6]
result = sum_of_even_numbers(numbers)
print("The sum of even numbers:",result)
```

```
The sum of even numbers: 12
```

2. Create a Python function that accepts a string and returns the reverse of that string

```
def reverse_string(input_string):
   return input_string[::-1]



text = "hello"
reversed_text = reverse_string(text)
print("Reversed string:", reversed_text)
```

```
Reversed string: olleh
```

3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.

```
def square_numbers(numbers):
   return [num ** 2 for num in numbers]
   numbers = [1, 2, 3, 4, 5,6]
squared_numbers = square_numbers(numbers)
print("Squared numbers:", squared_numbers)
```

```
Squared numbers: [1, 4, 9, 16, 25, 36]
```

4. Write a Python function that checks if a given number is prime or not from 1 to 200.

```
def is_prime(number):
    if number < 2:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False


for num in range(1, 201):
    if is_prime(num):
        print(f"{num} is a prime number")


 print(f"{num} is a prime number")
```

```
200 is a prime number
```

```
primes_up_to_200()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-9778f3c466de> in <cell line: 1>()
----> 1 primes_up_to_200()

NameError: name 'primes_up_to_200' is not defined
```

```
def primes_up_to_200():
    """
    Prints all prime numbers from 1 to 200.
    """
    for num in range(1, 201):
        if is_prime(num):
            print(f"{num} is a prime number")
print(primes_up_to_200())
```

```
None
```

. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

```
class FibonacciIterator:
    """
    An iterator class to generate the Fibonacci sequence up to a specified number of terms.
    """
    def __init__(self, num_terms):
        """
        Initializes the iterator with the number of terms.

        Parameters:
            num_terms (int): The number of terms in the Fibonacci sequence.
        """
        self.num_terms = num_terms
        self.current_term = 0
        self.a = 0
        self.b = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.current_term >= self.num_terms:
            raise StopIteration
        if self.current_term == 0:
            self.current_term += 1
            return self.a
        elif self.current_term == 1:
            self.current_term += 1
            return self.b
        else:
            self.current_term += 1
            next_value = self.a + self.b
            self.a, self.b = self.b, next_value
            return next_value

# Example usage
num_terms = 10
fib_iterator = FibonacciIterator(num_terms)

print(f"First {num_terms} terms of the Fibonacci sequence:")
for fib in fib_iterator:
    print(fib)
```

```
First 10 terms of the Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
```

6. Write a generator function in Python that yields the powers of 2 up to a given exponent

```
def powers_of_two(max_exponent):
    """
    A generator that yields the powers of 2 up to a given exponent.

    Parameters:
        max_exponent (int): The maximum exponent.

    Yields:
        int: Powers of 2 from 2^0 to 2^max_exponent.
    """
    for exponent in range(max_exponent + 1):
        yield 2 ** exponent

# Example usage
max_exponent = 5
print(f"Powers of 2 up to 2^{max_exponent}:")
for power in powers_of_two(max_exponent):
    print(power)
```

```
Powers of 2 up to 2^5:
1
2
4
8
16
32
```

7. Implement a generator function that reads a file line by line and yields each line as a string

```
def read_file_line_by_line(file_path):
   with open(file_path, 'r') as file:
        for line in file:
            yield line
```

8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.

```
# List of tuples
tuples_list = [(1, 3), (4, 1), (2, 2), (5, 0)]

# Sort the list using a lambda function as the key
sorted_list = sorted(tuples_list, key=lambda x: x[1])

# Print the sorted list
print("Sorted list based on the second element:", sorted_list)
```

```
Sorted list based on the second element: [(5, 0), (4, 1), (2, 2), (1, 3)]
```

9. Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit

```
# Function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

# List of temperatures in Celsius
temperatures_celsius = [0, 20, 37, 100]

# Use map() to apply the conversion function to each temperature
temperatures_fahrenheit = list(map(celsius_to_fahrenheit, temperatures_celsius))

# Print the converted temperatures
print("Temperatures in Fahrenheit:", temperatures_fahrenheit)
```

```
Temperatures in Fahrenheit: [32.0, 68.0, 98.6, 212.0]
```

10. Create a Python program that uses `filter()` to remove all the vowels from a given string.

```
# Function to check if a character is not a vowel
def is_not_vowel(char):
    vowels = "aeiouAEIOU"
    return char not in vowels

# Input string
input_string = "Hello, how are you?"

# Use filter() to remove vowels
filtered_string = "".join(filter(is_not_vowel, input_string))

# Print the result
print("String without vowels:", filtered_string)
```

```
String without vowels: Hll, hw r y?
```

Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

Write a Python program, which returns a list with 2-tuples. Each tuple consists of the order number and the product of the price per item and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €.

Write a Python program using lambda and map.

```
# Sample list of orders: [order_number, price_per_item, quantity]
orders = [
    [34587, 4.50, 4],
    [98762, 10.00, 2],
    [77226, 25.00, 6],
    [88112, 7.25, 3]
]

# Use map with a lambda function to compute the required tuples
result = list(map(lambda order: (
    order[0],
    (order[1] * order[2]) + 10 if (order[1] * order[2]) < 100 else order[1] * order[2]
), orders))

# Print the result
print("Processed orders:", result)
```

```
Processed orders: [(34587, 28.0), (98762, 30.0), (77226, 150.0), (88112, 31.75)]
```

11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list:[47,11,42,13];
    17c8374d-069e-45b8-906e-a9dbbe15b9d3.jpg