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

Ans:-In Python, methods and functions have similar purposes but differ in important ways. Functions are independent blocks of code that can be called from anywhere, while methods are tied to objects or classes and need an object or class instance to be invoked.

Function:

A function is a block of reusable code designed to perform a specific task.

Functions can be defined using the def keyword and are not inherently bound to any object or class.

Functions can be standalone or part of a module or a script.

They are called by their name and passing the required arguments, like this: 'function_name(arguments)'.

Method:

A method is a function that is associated with an object or class. In other words, methods are functions that are defined within a class and are meant to operate on instances of that class (or the class itself, in the case of class methods).

Methods are called on an object or class, and they implicitly take the object (or class) as their first argument, which is typically referred to as self for instance methods and cls for class methods.

```
#Example of a function:


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

print(greet("Alice"))
```

```
Hello, Alice!
```

```
#Example of an method:

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

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

g = Greeter("Bob")
print(g.greet())
```

```
Hello, Bob!
```

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

Ans:

#Parameters:

.Definition: Parameters are the variables listed in the function definition. They act as placeholders for the values that will be passed to the function when it is called.

.Purpose: They define what kind of input the function expects and how the function will use these inputs.

#Arguments:

.Definition: Arguments are the actual values or data you pass to the function when you call it. They replace the parameters in the function definition.

.Purpose: They provide the actual data that the function will operate on.

```
# Example of parameter:

def add_func(a,b):
    sum = a + b
    return sum
num1 = int(input("Enter the value of the first number: "))
num2 = int(input("Enter the value of the second number: "))
print("Sum of two numbers: ",add_func(num1, num2))
```

```
Enter the value of the first number: 56
Enter the value of the second number: 90
Sum of two numbers:  146
```

```
#Exmaple of argument

def details(name, age, grade):
    print("Details of student:", name)
    print("age: ", age)
    print("grade: ", grade)
details("Raghav", 12, 6)
details("Santhosh", grade = 6, age = 12)
```

```
Details of student: Raghav
age:  12
grade:  6
Details of student: Santhosh
age:  12
grade:  6
```

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

Ans:

The four steps to defining a function in Python are the following:

.Use the keyword def to declare the function and follow this up with the function name.

.Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.

.Add statements that the functions should execute.

.End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

```
#Example:

def hello():
  name = str(input("Enter your name: "))
  if name:
    print ("Hello " + str(name))
  else:
    print("Hello World")
  return

hello()

Enter your name: Rahil
Hello Rahil
```

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

Ans:

The Python return statement marks the end of a function and specifies the value or values to pass back from the function. Return statements can return data of any type, including integers, floats, strings, lists, dictionaries, and even other functions.

```
#Example:

def calculate_area(length, width):
     area = length * width
     return area

area = calculate_area(50, 50) # get the result of the function
print(f"Calculated area: {area}") # use the result of the function

Calculated area: 2500
```

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

Ans:

Iterables:

Definition: An iterable is any Python object capable of returning its members one at a time, allowing iteration over its contents. In essence, iterables are objects that can be looped over (e.g., using a for loop).

#Characteristics:

They implement the **iter**() method, which returns an iterator object. Common examples of iterables include lists, tuples, strings, dictionaries, and sets.

Iterators :

Definition: An iterator is an object that represents a stream of data. It returns the next value when you call the **next**() method and keeps track of its state to know where it is in the iteration process.

#Characteristics:

They implement two methods: **iter**() (which returns the iterator object itself) and **next**() (which returns the next value in the sequence). Iterators are used to control the iteration process and can be explicitly created from iterables.

```
#Example:


# A list is an iterable
numbers = [1, 2, 3, 4]

# You can iterate over it using a for loop
for number in numbers:
    print(number)
```

```
1
2
3
4
```

```
#Example:


# Create an iterator from a list
numbers = [1, 2, 3, 4]
iterator = iter(numbers)

# Iterate through the iterator
print(next(iterator))
print(next(iterator))
```

```
1
2
```

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

Ans:

- Generators in Python are a special type of iterator that allows you to iterate over a sequence of values, but unlike lists or other iterables, they generate values on-the-fly and only when needed. This makes them more memory-efficient, especially when dealing with large data sets or streams of data.

1. Generator Functions:

- Definition: Generator functions use the yield keyword instead of return. When the yield statement is encountered, the function's state is saved, and the value is returned to the caller. The function can then be resumed from where it left off the next time next() is called.

```
#Example:


def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
for num in counter:
    print(num)

1
2
3
4
5
```

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

Ans:

1. Memory Efficiency:

   Lazy Evaluation: Generators produce values one at a time and only when needed. This means they don't require storing all values in memory at once, which is especially useful for working with large datasets or infinite sequences.

  Reduced Memory Footprint: Since generators only generate values on-the-fly, they use significantly less memory compared to storing large amounts of data in a list or other data structures.

1. Performance:

On-Demand Computation: Generators compute values as they are requested, which can be more efficient than computing all values upfront, especially if only a subset of the values is needed.

Faster Startup: Generators start producing results immediately without waiting for the entire sequence to be generated, which can improve the responsiveness of the application

1. Infinite Sequences :

Handling Infinite Sequences: Generators are ideal for generating potentially infinite sequences of data. For example, you can create a generator for an infinite sequence of Fibonacci numbers without running out of memory or requiring an enormous data structure.

Continuous Data Streams: They can be used to handle continuous data streams, such as reading large files line-by-line or processing real-time data feeds.

1. Simplified Code :

Concise and Readable: Generators often lead to more concise and readable code. For example, using a generator expression can be simpler and more elegant than equivalent list comprehensions when the result is only needed one item at a time.

State Management: Generators inherently manage their state between yields, so there's no need for additional bookkeeping in the code to keep track of the state of iteration.

1. Integration with Iteration Protocol :

Built-In Iteration Support: Generators implement the iterator protocol (**iter**() and **next**()), making them compatible with Python's iteration constructs, such as for loops and functions that consume iterables (e.g., sum(), list()).

1. Composability:

Pipeline of Generators: Generators can be composed to form pipelines. For instance, you can chain multiple generators together to process data in stages without creating intermediate data structures.

Modular Functions: Each generator function can perform a specific transformation or filtering step, making the code modular and easier to maintain.

```python
#Memory Efficiency:

def large_range(n):
    for i in range(n):
        yield i

gen = large_range(10**6)

#Performance:

def countdown(n):
    while n > 0:
        yield n
        n -= 1
for number in countdown(5):
    print(number)

#Infinite Sequences:

def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
fib = fibonacci()
for _ in range(10):
    print(next(fib))

#Simplified Code:

squares = (x * x for x in range(10))
for square in squares:
    print(square)

#Composability:


def even_numbers(n):
    for i in range(n):
        if i % 2 == 0:
            yield i
def squared_numbers(iterable):
    for x in iterable:
        yield x * x

# Chaining generators
evens = even_numbers(10)
squared_evens = squared_numbers(evens)
for num in squared_evens:
    print(num)
```

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

Ans:

A lambda function in Python is a small, anonymous function defined using the lambda keyword. Unlike regular functions created with the def keyword, lambda functions are typically used for simple, short-lived operations. Here’s a detailed look at lambda functions and their typical use cases:

Typical Use Cases

Short-term Use in Higher-Order Functions:

Lambda functions are often used as arguments to higher-order functions like map(), filter(), and sorted(). These functions apply a given function to a sequence or iterable.

```
# Examples
#Using Lambda with map()

numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))
# squares: [1, 4, 9, 16, 25]

#Example
#Using Lambda with filter()

numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
# even_numbers: [2, 4]

#Example
#Using Lambda for Sorting

data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
sorted_data = sorted(data, key=lambda item: item[1])
# sorted_data: [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
```

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

Ans:

The map() function in Python is a built-in function used to apply a specified function to each item in an iterable (such as a list, tuple, or set) and return a map object (an iterator) of the results. It is a key component of functional programming in Python, enabling the transformation of data structures in a clean and efficient way.

```
#Example:

# List of temperatures in Celsius
celsius_temperatures = [0, 10, 20, 30, 40]

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

# Use map() to apply the conversion function to each element in the
list
fahrenheit_temperatures = map(celsius_to_fahrenheit,
celsius_temperatures)

# Convert the map object to a list to view the results
fahrenheit_temperatures = list(fahrenheit_temperatures)
```

```
# Print the converted temperatures
print(fahrenheit_temperatures)

[32.0, 50.0, 68.0, 86.0, 104.0]
```

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

Ans:

The functions 'map()', 'reduce()', and 'filter()' in Python are all built-in functions used for processing iterables and are key components of the functional programming paradigm. Each serves a different purpose:

1. map()

   Purpose: Applies a given function to each item of an iterable and returns an iterator of the results.

2.. reduce()

Purpose: Applies a function cumulatively to the items of an iterable, reducing it to a single value.

Use Case: When you want to perform cumulative operations like summing all elements, finding the product, etc.

1. filter()

   Purpose: Filters elements of an iterable for which a function returns True.

Use Case: When you want to extract elements that meet certain criteria from an iterable.

```
#Example map():

# Convert temperatures from Celsius to Fahrenheit
celsius = [0, 10, 20, 30]
fahrenheit = map(lambda x: (x * 9/5) + 32, celsius)
print(list(fahrenheit))

[32.0, 50.0, 68.0, 86.0]

#Example reduce():

from functools import reduce

# Calculate the product of a list of numbers
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)

24
```

```python
#Example filter():

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

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

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

Ans:

It is in google drive link