Q-1) What is the difference between a function and a method in Python?

Ans) Function:

- A block of code that performs a task.
- It is an independent function.
- It is independent and not tied to any specific object.
- We use def keyword to define a function.
- After defining the function, we also have to call this using function name.
- Argument and parameters can be passed to a function.

Example:

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

print(greet("Alok"))
```

Method:

- A function associated with an object (like strings, lists, or custom objects).
- You call it on an object, and it often acts on that object.
- we can access method using dot (.) operator .
- There are different methods for list, tuple, string, sets and dictionary.

Example :-

```
name = "Alok"
print(name.upper())
print(name.capitalize())
print((name.lower())
```

➔ This are the example of methods.

Q-2) Explain the concept of function arguments and parameters in Python.

Ans) Paramters and arguments are related to function.

Paramters :-

➔ Parameters are placeholders for values that a function accepts when it is defined.

➔ They act like variables inside the function to process the data you pass when calling the function.

Arguments :-

-> Arguments are the actual values you pass to a function when calling it.

-> These values are substituted for the parameters during the function call.

➔ Parameters are defined in the function definition.

➔ Arguments are used in the function call.

Types of Arguments :-

- Positional argument
- Keyword argument
- Arbitary argument
- Default argument

Example :-

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

print(add(5, 3))        # positional arguments

print(add(b=3, a=5))    # keyword argument

def greet(name="Guest"):    # default argument
    return f"Hello, {name}!"

print(greet())
print(greet("Alok"))
```

Q-3) What are the different ways to define and call a function in Python?

Ans) In Python, there are several ways to define and call functions.

1. Standard Function Definition and Call

➔ Use the def keyword to define a function.

➔ Use the function name followed by parentheses.

Example:

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

print(greet("Alok"))
```

2. Function with Default Parameters

-> You can assign default values to parameters.

➔ If no argument is provided, the default value is used.

Example:

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

print(greet())
print(greet("Alok"))
```

3. Function with Variable-Length Arguments :

➔ *args (Positional Arguments): Accepts multiple arguments as a tuple.

➔ **kwargs (Keyword Arguments): Accepts multiple named arguments as a dictionary.

Example:

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3, 4))

def display_info(**kwargs):
    return kwargs

print(display_info(name="Alok", age=20))
```

4. Lambda Function (Anonymous Function) :-

➔ A one-line function using the lambda keyword.

➔ Assign it to a variable or use it directly.

Example:

```
square = lambda x: x * x
print(square(5))

print((lambda x, y: x + y)(3, 7))
```

5. Function with Return Values :

➔ Use the return keyword to send back results.
➔ Capture the return value in a variable or use it directly.

Example:

```
def multiply(a, b):
    return a * b

result = multiply(4, 5)
print(result)
```

6. Recursive Function :

➔ A function that calls itself.
➔ Provide an argument that eventually stops recursion.

Example:

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

Q-4) What is the purpose of the `return` statement in a Python function?

Ans)

- ➔ The return statement in a Python function is used to send a result or value back to the part of the program where the function was called.
- ➔ It essentially ends the function execution and "returns" the specified value to the caller.
- ➔ It allows the function to produce a result that can be used elsewhere in the program.

Example :-

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

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

- ➔ Once the return statement is executed, the function stops, and no further code in the function is executed.
- ➔ The return statement can be used with conditions to return specific results based on the logic.
- ➔ If no return statement is used, or it is written as return without a value, the function returns None by default.
- ➔ Python allows a function to return multiple values as a tuple.

```
def example():
    print("Before return")
    return "Value"
    print("After return")

print(example())
```

Q-5) What are iterators in Python and how do they differ from iterables?

Ans) Iterators and iterables are both used for looping (iterating) in Python.

- Iterables :-

➔ An iterable is any object that can be looped over using a for loop.

➔ Examples: Lists, tuples, strings, dictionaries, sets, etc.

➔ Iterable objects have a method called __iter__() that allows you to create an iterator from them.

Example :-

```
my_list = [1, 2, 3]
for item in my_list:
    print(item)
```

- Iterators :-

- ➔ An iterator is an object that represents a stream of data, producing one value at a time when you call next() on it.
- ➔ Iterators are created from iterables using the iter() function.
- ➔ They keep their state during iteration (i.e., they remember where they left off).

Examples :-

```
my_list = [1, 2, 3]
iterator = iter(my_list)

print(next(iterator))
print(next(iterator))
print(next(iterator))
```

Q-6) Explain the concept of generators in Python and how they are defined.

Ans)

- ➔ A generator in Python is a special type of function that allows you to produce a sequence of values one at a time, as they are needed.
- ➔ Instead of producing all the values at once and storing them in memory. Generators are very memory-efficient for handling large data streams.
- ➔ A regular function uses return to return a single value and then exits.
- ➔ A generator uses the yield keyword to produce a value, pausing the function without terminating it, so it can resume execution where it left off.
- ➔ Every time yield is called, the generator pauses execution and keeps track of its state for resumption.

➔ Generators are defined like regular functions but include one or more yield statements.

Example :-

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
for _ in range(10):
    print(next(fib))
```

Q-7) What are the advantages of using generators over regular functions?

Ans)

➔ Generators offer several advantages over regular functions, especially when working with large datasets or streams of data.

➔ These advantages stem from their one evaluation at a time and state retention capabilities.

➔ Generators do not store all the values in memory at once. Instead, they produce values on-demand, one at a time.

➔ This makes generators ideal for handling large datasets or infinite sequences.

➔ Generators produce values only when they are needed, delaying computations until the results are requested.

➔ This reduces computation time when not all values are required.

➔ With regular functions, infinite sequences would require complex implementations. Generators handle them naturally.

➔ Generators automatically remember their state between calls to next(). This eliminates the need for maintaining external variables to track progress.

➔ Generators simplify the implementation of iterators. Instead of manually defining __iter__() and __next__() methods, you can use yield to produce values on-the-fly.

Example :-

```
def infinite_counter(start=0):
    while True:
        yield start
        start += 1

counter = infinite_counter()
print(next(counter))
print(next(counter))
```

Q-8) What is a lambda function in Python and when is it typically used?

Ans)

➔ A lambda function is a small, anonymous (nameless) function in Python.

➔ It is defined using the lambda keyword and can have any number of arguments but only one expression.

➔ The result of the expression is automatically returned.

- Syntax :

lambda arguments: expression

Example :-

```
add = lambda x, y: x + y
print(add(3, 7))
```

- ➔ Lambda functions are used for operations that don't require a full function definition.
- ➔ With Functions Like map(), filter(), and reduce()
- ➔ These functions often require another function as input, and lambda is ideal for defining quick, one-off functions.

Example with map( ) :-

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

Example with filter( ) :-

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

Example with reduce( ) :-

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

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

Ans)

- ➔ The map() function in Python is used to apply a given function to each item in an iterable (such as a list, tuple, etc.)
- ➔ And return an iterator that yields the results. This allows you to perform operations on all elements of an iterable without having to use explicit loops, making the code more concise and readable.

• Syntax :

map(function, iterable, ...)

- ➔ The function is applied to each element of the iterable.
- ➔ If there are multiple iterables, the function will receive one item from each iterable.
- ➔ The result of applying the function is returned as an iterator.

Example :-

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

    # Define a function that squares a number
    def square(x):
        return x * x

    # Apply the square function to each element in the list
    squared_numbers = map(square, numbers)

    # Convert the map object to a list to see the result
    print(list(squared_numbers))
```

➔ If you pass multiple iterables to map(), the function will receive one element from each iterable and apply the function to those elements in parallel.
Example :-

```
list1 = [1, 2, 3, 4]
list2 = [10, 20, 30, 40]

# Define a function to add two numbers
def add(x, y):
    return x + y

# Apply the add function to elements from both lists
sum_lists = map(add, list1, list2)

print(list(sum_lists))
```

Q-10) What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

Ans)

➔ The map(), reduce(), and filter() functions in Python are all part of functional programming tools that allow you to process iterables in a concise and efficient way.

➔ Each of these functions serves a different purpose,

• map() Function :-

➔ Purpose: map() applies a function to every element in an iterable (or multiple iterables) and returns an iterator that produces the results.

➔ Use Case: When you want to transform each element of an iterable based on some function.

Example :-

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

- reduce() Function :-

➔ Purpose: reduce() applies a binary function (a function that takes two arguments) cumulatively to the items of an iterable, reducing the iterable to a single value. It repeatedly applies the function to pairs of values until only one result remains.

➔ Use Case: When you need to accumulate or combine the items in an iterable into a single result (like summing up elements, multiplying them, etc.).

Example :-

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

- filter() Function :-

➔ Purpose: filter() applies a function to each item in an iterable and returns an iterator that contains only those items for which the function returns True.

➔ Use Case: When you want to filter out elements from an iterable based on a condition (i.e., select items that satisfy a particular condition).

Example :-

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

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

✱ Sum Operation Using reduce() function

Given List :-

[47, 11, 42, 13]

function :-

lambda x, y : x + y (adds two no. together)

Step-by-Step Process :-

[47, 11, 42, 13]

↳ ↙
x + y

1) First iteration :-

Take the first two element : 47 and 11

Step:1

Apply the lambda function :

47 + 11 = 58

Intermediate result : 58

2) Second iteration :-

Step : 2 [47, 11, 42, 13]

58 + 42
x y

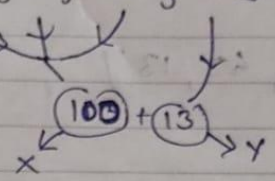
Apply The lambda func :-

58 + 42 = 100

Intermediate result : 100

3) Third iteration :-

Step : 3 [47, 11, 42, 13]



Applying the lambda function :-

100 + 13 = 113

Final result : 113

Final Output :

```
from functools import reduce
num = [47, 11, 42, 13]
result = reduce (lambda x, y : x + y , num)
print (result)
```