Note: For each theory Question, give at least one example.

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

Ans. In Python, a function and a method are both blocks of code that can be executed multiple times from different parts of a program. However, there are key differences between them:

Function:

- A function is a self-contained block of code that takes arguments, performs some operations, and returns a result.

- Functions are defined using the def keyword.

- They can be called independently, passing required arguments.

Method:

- A method is a function that is bound to an object or a class.

- It is called on an instance of a class (an object) and has access to the object's attributes and other methods.

- Methods are also defined using the def keyword, but inside a class definite

To illustrate the difference, consider a simple example:

Function

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

Method

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print(f"Hello, {self.name}!")
```

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

Ans. In Python, functions can take inputs, known as arguments, which are values passed to the function when it's called. These arguments are assigned to variables, known as parameters, defined in the function definition.

Parameters:

- Are the variables listed in the function definition.

- Are used to receive the input values passed to the function.

Arguments:

- Are the actual values passed to the function when it's called.

- Can be literals, variables, or expressions.

Here's an example:

```
def greet(name, message):  name and message are parameters
    print(f"{message}, {name}!")
greet("John", "Hello")   "John" and "Hello" are arguments
```

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

ANS. Defining Functions

1. Standard Function Definition

- def function_name(parameters):

- Example: def greet(name): print(f"Hello, {name}!")

2. Lambda Functions (anonymous functions)

- lambda parameters: expression

- Example: greet = lambda name: print(f"Hello, {name}!")

3. Nested Functions (functions inside functions)

- Example: def outer(): def inner(): print("Hello"); inner()

Calling Functions

1. Positional Argument

- function_name(arg1, arg2, ...)

- Example: greet("John")

2. Keyword Arguments

- function_name(param1=value1, param2=value2, ...)

- Example: greet(name="John")

3. Default Argument Values

- function_name(param1=value1, param2=value2, ...)

- Example: def greet(name="World"): print(f"Hello, {name}!"); greet()

4. Variable-Length Arguments (*args and **kwargs)

- function_name(*args, **kwargs)

- Example: def greet(*names): for name in names: print(f"Hello, {name}!"); greet("John", "Jane", "Bob")

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

ANS. The return statement in a Python function:

- Exits the function and passes control back to the caller

- Returns values to the caller (single or multiple values)

In short, return controls the function's output and exit point.

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

ANS. Iterables:

- Are objects that can be iterated over (e.g., lists, tuples, dictionaries, sets, strings)

- Have an __iter__ method that returns an iterator object

- Can be used in loops, comprehensions, and other iteration contexts

Iterators:

- Are objects that keep track of their position in an iterable

- Have a __next__ method that returns the next value in the iteration

- Can only be iterated over once; then they're exhausted

- Are created by calling the __iter__ method on an iterable

Key differences:

- Iterables can be iterated over multiple times; iterators can only be used once.

- Iterables have an __iter__ method; iterators have a __next__ method.

Example:

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

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

ANS. In Python, a generator is a special type of iterable object that can be used to generate a sequence of values , instead of storing them in memory all at once. Generators are defined using a function with the yield keyword.

Here's a simple example:

```
def sq_num_gen(n):
    for I in range (n):
        yield i**2
gen =sq_num_gen(10)
   next(gen)
```

To define a generator:

1. Write a function.

2. Use the yield keyword instead of return.

3. The function will return a generator object.

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

Ans. Advantages of using generators over regular functions:

]1. Memory Efficiency: Generators use significantly less memory, as they only store the current value and state, whereas regular functions store the entire sequence in memory.

2. Lazy Evaluation: Generators compute values on-the-fly, only when needed, whereas regular functions compute all values upfront.

3. Flexibility: Generators can be used to create infinite sequences, whereas regular functions cannot.

4. Improved Performance: Generators can perform better, as they avoid the overhead of creating and storing large datasets.

5. Simplified Code: Generators can simplify code, as they eliminate the need for manual iteration and indexing.

Overall, generators provide a powerful and efficient way to work with sequences of data in Python, especially when dealing with large datasets or infinite sequences.

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

Ans. A lambda function in Python is a small, anonymous function that can take any number of arguments, but can only have one expression. It's typically used when:

1. A simple function is needed, and defining a full function with def is unnecessary.

2. A function needs to be passed as an argument to another function.

3. A function needs to be returned from another function.

4. A quick, one-time-use function is needed.

Lambda functions are defined using the lambda keyword, followed by the arguments, a colon, and the expression. Example:

double = lambda x: x * 2

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

ANS. The purpose of map() is to:

1. Transform data: Apply a function to each item in an iterable to transform or modify the data.

2. Simplify code: Instead of writing a loop to apply a function to each item, map() provides a concise way to achieve the same result.

Usage:

map(function, iterable)

- function: The function to apply to each item in the iterable.

- iterable: The list, tuple, string, or other iterable to which the function is applied.

Example:

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

squared_numbers = list(map(lambda x: x**2, numbers))

print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python? (Attach paper image for this answer) in doc or colab notebook.

ANS. 1. map():

- Applies a given function to each item of an iterable (like a list, tuple, or string) and returns a map object.

- Purpose: Transform data by applying a function to each item.

- Syntax: map(function, iterable)

2. reduce() :

- Applies a given function to the first two items of an iterable, then to the result and the next item, and so on, reducing the iterable to a single output.

- Purpose: Aggregate or combine data by applying a function cumulatively.

- Syntax: reduce(function, iterable)

3. filter():

- Applies a given function to each item of an iterable and returns a filter object containing only the items for which the function returns True.

- Purpose: Select or filter data based on a condition.

- Syntax: filter(function, iterable)

Example usage:

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

- map(): squared_numbers = list(map(lambda x: x**2, numbers))

- reduce(): sum_of_numbers = reduce(lambda x, y: x + y, numbers)

- filter(): even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

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

ANS. numbers = [47, 11, 42, 13]

reduce() applies the given function (in this case, addition) to the first two elements, then to the result and the next element, and so on.

Here's the step-by-step process:

1. reduce() starts with the first two elements: 47 and 11

   - 47 + 11 = 58 (initial result)

2. reduce() applies the function to the result (58) and the next element (42):

   - 58 + 42 = 100 (new result)

3. reduce() applies the function to the result (100) and the next element (13):

   - 100 + 13 = 113 (final result)

So, the final output of reduce(lambda x, y: x + y, numbers) would be 113.