Functions -- Assignment

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

Sol - Function :-
A function is a block of code that performs a specific task, It is
Called independently using its name and it is Defined outside of
classes.
Example -- len([1, 2, 3]) → len is a built-in function.

Method :-
A method is a function that is associated with an object (belongs to a
class) and is Called using an object (e.g., object.method()) also
Defined inside a class.
Example -- "hello".upper() → upper is a method of the str class.

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

Sol - In Python, parameters and arguments are distinct but related
concepts concerning how data is passed into functions.
Parameters:
Parameters are the variables defined within the parentheses of a
function's def statement.
They act as placeholders for the values that the function expects to
receive when it is called.
Parameters define the type and number of inputs a function can accept,
forming part of its signature.
They are local to the function where they are defined.
Example of Parameters:
def greet(name):  # 'name' is a parameter
    print(f"Hello, {name}!")

Arguments:
Arguments are the actual values passed to a function when it is
called.
They fill the placeholders (parameters) during the function's
execution.
Arguments can be literals, variables, or expressions.
Example -
user_name = "Alice"
greet(user_name)  # 'user_name' is an argument
greet("Bob")      # "Bob" is an argument

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

Sol - In Python, there are several ways to define and call functions,
```

```
depending on your needs
from simple reusable functions to ones with arguments, default values,
or even anonymous (lambda) functions.
1. Basic Function Definition and Call
def greet():
    print("Hello, World!")

greet()   # Function call
2.  Function with Parameters
def greet(name):
    print("Hello,", name)

greet("Nikhil")  # Argument passed
3. Function with Default Parameters
def greet(name="Guest"):
    print("Hello,", name)

greet()          # Uses default
greet("Nikhil")  # Overrides default
4. Lambda (Anonymous) Function
square = lambda x: x * x
print(square(5))

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

Sol - The return statement in Python is used to send a value back from
a function to the part of the program that called it.
Purpose of return

1.Gives Output Back to Caller
A function can process data and then return a result.
Without return, the function just runs but gives no usable output.

2.Ends Function Execution
Once return is executed, the function stops running immediately.

3.Allows Reusability of Results
You can store the returned value in a variable for further use.

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

Sol - An iterable is any Python object you can loop over using a for
loop.
It's something that can give you an iterator when you call iter() on
it.
Examples of iterables:

Lists → [1, 2, 3]
```

Tuples → (4, 5, 6)

Strings → "Hello"

Dictionaries → {'a': 1, 'b': 2}

Sets → {1, 2, 3}

An iterator is an object that keeps track of where you are in a sequence and gives you the next item when you call next().

It is created from an iterable using the iter() function.

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

sol - A generator is a special type of iterator that allows you to generate values one at a time, instead of storing them all in memory at once.

They are used for lazy evaluation meaning they produce items only when needed.
here are two main ways to define generators in Python:

1. Using a Generator Function

A function that uses the keyword yield instead of return.
Example -

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count  # returns a value, but remembers where it left off
        count += 1

# Using the generator
for num in count_up_to(5):
    print(num)
```

Using Generator Expressions

A shorter, one-line version of a generator (similar to list comprehension but with parentheses).
Example -

```
squares = (x * x for x in range(5))
print(next(squares))  # 0
print(next(squares))  # 1
```

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

Sol - Memory efficiency: Generators create values one at a time and don't need to store the entire sequence in memory.
 This makes them significantly more memory-efficient, especially for large datasets.
 Lazy evaluation: Generators use lazy evaluation, meaning they only compute a value when it's requested.
 This can improve performance because the program doesn't have to do all the work upfront.
 Infinite sequences: Generators can represent infinite sequences, such as the Fibonacci sequence, because they produce values on the fly.
 A regular function cannot return an infinite sequence because it would be impossible to store it all in memory.
 Pipelining: Multiple generators can be chained together to create data processing pipelines.
 Simpler code: Generators pause their execution with yield and can be resumed later. This allows for writing more readable and maintainable code for iterative algorithms compared to managing state in a regular function.

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

Sol - A lambda function in Python is a small, anonymous function defined using the lambda keyword, rather than the def keyword used for regular functions.
It's anonymous because it doesn't require a name. Lambda functions are restricted to a single expression, which is implicitly returned. They can take any number of arguments.
Syntax -- lambda arguments: expression

It is typically used when one-time operations, especially in conjunction with higher-order functions that accept other functions as arguments. Common scenarios include:
With map(), filter(), and sorted(): Lambda functions provide a concise way to define the transformation, filtering condition, or sorting key directly within these functions.As a simple callback function: When a function requires a small, custom operation to be performed, a lambda can be passed as a callback.
For creating small, throwaway functions: When a function is needed only for a specific, isolated task and doesn't warrant a formal def definition.
While lambda functions offer conciseness, for more complex or multi-line logic, a regular def function is generally preferred for readability and maintainability.

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

Sol - The map() function in Python serves to apply a given function to each item in an iterable (such as a list, tuple, or set) and returns

an iterator that yields the results. This provides a concise and efficient way to perform transformations on collections of data.
Purpose:
The primary purpose of map() is to streamline the process of applying a function to every element of an iterable without the need for explicit loops, leading to more readable and often more performant code, especially for large datasets. It embodies a functional programming paradigm, emphasizing the transformation of data.
Usage:
The syntax for map() is: map(function, iterable, ...)
function: This is the function that will be applied to each item of the iterable(s). It can be a built-in function, a user-defined function, or a lambda function.
iterable: This is one or more iterable objects whose elements will be passed as arguments to the function. If multiple iterables are provided, the function must accept a corresponding number of arguments.

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

Sol - The Python functions map(), filter(), and reduce() are all higher-order functions that operate on iterables, but they serve distinct purposes:
1. map():
Purpose: Applies a given function to each item of an iterable and returns an iterator that yields the results. It transforms each element individually.
Output: An iterable (map object) of the same length as the input, containing the transformed elements.
Example:

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

2. filter():
Purpose: Constructs an iterator from elements of an iterable for which a function returns true. It selects elements based on a condition.
Output: An iterable (filter object) containing only the elements for which the applied function returned True. The length of the output can be less than or equal to the input.
Example:

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

3. reduce():
Purpose: Applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value. (Note: reduce() is in the functools module in Python 3,

```
not a built-in function.)
Output: A single value resulting from the cumulative application of
the function.
Example:
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

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

Sol -

Solution –

Given :-

List → [47, 11, 42, 13]

we want to sum all elements using reduce() function.

Step1 : Import And Define.

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

Output – 113.

Step2 : Internal Mechanism

reduce(function, iterable) works by :-

Taking 1st two elements → applying the function – then applying the same fun. on result & the next element

Step	Operation	Result
1	47 + 11	58
2	58 + 42	100
3	100 + 13	113

If we write manually reduce() essentially does:

result = (((47+11)+42)+13)

So, the lambda function (lambda x,y: x+y) is called 3 times for a list of 4 elements.