Functions Assignment

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

In Python, the terms function and method both refer to callable objects, but they are used in different contexts and have distinct characteristics.

(i) Definition :
> Function: A function is a block of code that is defined to perform a specific task. It can be defined using the def keyword and can be called by its name.
> Method: A method is essentially a function that is associated with an object (typically an instance of a class).
Method operates on the data contained within that object and is defined within a class.

(ii) Context :
> Function: Functions are defined at the module level or globally and are not tied to any particular object or class.
> Method: Methods are tied to the objects or classes and are typically invoked on an object (or class) instance.

(iii) Invocation :
> Function: Functions are called directly by their name, and you pass the necessary arguments.

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

greet("Alice")  # Calling a function
```

> Method: Methods are called on an instance of a class (or a class itself if it is a class method). The first argument of a method is usually self, which refers to the instance of the object.

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

person = Person()
person.greet("Alice")  # Calling a method
```

(iv) Binding :
Function: Functions are not bound to any specific object. They operate independently.
Method: Methods are bound to objects and can access or modify the object's internal data (attributes).

(v) Key Points :
Function: Independent and can be called directly.
Method: Associated with an object or class and is called on that object/class.

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

In Python, function arguments and parameters are essential concepts that allow you to pass data into functions, enabling functions to operate on that data.

(i) Function Parameters :
> Definition: Parameters are the variables that are defined in the function signature. These variables act as placeholders for the values that will be passed to the function when it is called.
> Where: Parameters are listed in the parentheses after the function name when the function is defined.

Example -
```
    def greet(name):  # 'name' is the parameter
        print(f"Hello, {name}!")
```
Here, name is the parameter of the greet function. It is used to receive the value passed during the function call.

(ii) Function Arguments :
> Definition: Arguments are the actual values or data you pass to a function when calling it. These values are assigned to the corresponding parameters in the function.
> Where: Arguments are passed in the function call.

Example -
```
    greet("Alice")  # "Alice" is the argument
```
In this example, "Alice" is the argument passed to the function greet. The value "Alice" is assigned to the name parameter of the function when it is executed.

(iii) Types of Function Arguments in Python :
Python supports different ways to pass arguments to functions, and they can be classified into several types.

(1) Positional Arguments :
These are arguments passed to the function in a specific order.
The first argument is assigned to the first parameter, the second argument to the second parameter, and so on.
Example -
```
def add(a, b):  # a and b are parameters
    return a + b

result = add(3, 5)  # 3 and 5 are positional arguments
print(result)  # Output: 8
```

(2) Keyword Arguments :

These are arguments passed to the function by explicitly specifying the parameter names. This allows you to pass arguments in any order.
Example -

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet(name="Alice", age=30)  # Using keyword arguments
greet(age=25, name="Bob")     # Order doesn't matter
```

(3) Default Arguments :
These are arguments that have a default value. If no value is provided during the function call, the default value is used.
Example -

```
def greet(name, age=25):  # 'age' has a default value of 25
    print(f"Hello {name}, you are {age} years old.")

greet("Alice")   # Uses default age = 25
greet("Bob", 30) # Overrides default age
```

(4) Variable - Length Arguments :
Sometimes, you might not know how many arguments you want to pass to a function. Python allows you to pass a variable number of arguments using *args and **kwargs.
*args: Allows you to pass a variable number of non-keyword arguments (positional).

Example of *args -

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

result = sum_numbers(1, 2, 3, 4)  # Pass any number of arguments
print(result)  # Output: 10
```

Example of **kwargs -

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)  # Passes key-value pairs
```

> Parameters are defined in the function signature and act as placeholders.
> Arguments are the actual values you pass to a function when calling it.
> Python allows flexible argument passing with positional, keyword, default, and variable-length arguments.

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

In Python, there are several ways to define and call a function. Below are the different ways to define and call functions in Python :

(i) Basic Function Definition and Call :
This is the most common way to define and call a function in Python.

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

Function Call :
```
greet("Alice")  # Calling the function with an argument
```

(ii) Function with Default Arguments :
You can define a function with default values for its parameters. If no argument is passed during the function call, the default value is used.

Function Definition :
```
def greet(name, age=25):  # age has a default value of 25
    print(f"Hello, {name}. You are {age} years old.")
```

Function Call :
```
greet("Alice")        # Uses default age = 25
greet("Bob", 30)      # Passes the age argument, overriding the default
```

(iii) Function with Variable-Length Arguments (*args) :
You can define a function that accepts a variable number of positional arguments using *args.

Function Definition :
```
def sum_numbers(*args):  # Accepts any number of positional arguments
    return sum(args)
```

Function Call :
```
print(sum_numbers(1, 2, 3))  # Output: 6
print(sum_numbers(4, 5, 6, 7, 8))  # Output: 30
```

(iv) Function with Keyword Arguments (**kwargs) :
You can define a function that accepts a variable number of keyword arguments using **kwargs.

Function Definition :
```
def print_info(**kwargs):  # Accepts any number of keyword arguments
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

Function Call :

```
print_info(name="Alice", age=30)
# Output:
# name: Alice
# age: 30
```

(v) Function with Both *args and **kwargs :
You can define a function that accepts both variable-length positional and keyword arguments.

Function Definition :

```
def display_info(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)
```

Function Call :

```
display_info(1, 2, 3, name="Alice", age=30)
# Output:
# Positional arguments: (1, 2, 3)
# Keyword arguments: {'name': 'Alice', 'age': 30}
```

(vi) Lambda Functions (Anonymous Functions) :
A lambda function is a small, anonymous function defined using the lambda keyword. It can have any number of parameters but only one expression.

Function Definition :

```
multiply = lambda x, y: x * y
```

Function Call :

```
print(multiply(3, 4))  # Output: 12
```

(vii) Function Assigned to a Variable (Function as an Object) :
Functions in Python are first-class citizens, meaning they can be assigned to variables or passed around as arguments to other functions.

Function Definition :

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

Function Call :

```
greeting = greet  # Assigning the function to a variable
print(greeting("Alice"))  # Output: Hello, Alice!
```

(viii) Recursive Functions :
A function can call itself. This is known as recursion. It is useful for problems that can be broken down into smaller subproblems (like calculating factorials or traversing trees).

Function Definition :
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Function Call :
```
print(factorial(5))  # Output: 120
```

(ix) Function with Multiple Return Values :
In Python, you can return multiple values from a function as a tuple.

Function Definition :
```
def get_min_max(values):
    return min(values), max(values)  # Returns a tuple (min, max)
```

Function Call :
```
min_val, max_val = get_min_max([1, 2, 3, 4, 5])
print(min_val, max_val)  # Output: 1 5
```

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

The return statement in Python is used to exit a function and send back a value to the caller. When the return statement is executed, the function terminates immediately, and the specified value is returned to the place where the function was called.

Key Purposes of the return Statement -

(i) Return a Value :
The primary purpose of return is to send a value back to the caller. This allows you to pass the result of a computation or an operation from the function to the code that called it.

Example -
```
def add(a, b):
    return a + b  # Return the sum of a and b

result = add(3, 5)  # Call the function and store the result in 'result'
```

```
print(result)  # Output: 8
```

(ii) End Function Execution :
Once a return statement is executed, the function immediately stops, and no further code inside the function is executed.

Example -
```
def greet(name):
    print("Hello, " + name)
    return  # Function terminates here
    print("This line will never be reached.")  # Unreachable code

greet("Alice")
```
In this example, the second print statement will never be executed because the function ends when the return statement is encountered.

(iii) Returning Multiple Values :
You can use return to send multiple values back to the caller. Python automatically packages these values into a tuple.

Example -
```
def get_min_max(values):
    return min(values), max(values)  # Returning two values as a tuple

min_val, max_val = get_min_max([1, 2, 3, 4, 5])
print(min_val, max_val)  # Output: 1 5
```

(iv) Return None :
If a function does not explicitly return a value, it implicitly returns None. This is useful when you don't need to return a result from the function.

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

result = greet("Alice")
print(result)  # Output: None
```

(v) Control Flow :
The return statement also provides a way to control the flow of a function. You can use it to exit early from the function based on a condition, avoiding unnecessary operations.

Example -
```
def check_even_or_odd(number):
    if number % 2 == 0:
        return "Even"  # Return early if number is even
    return "Odd"  # Return if number is odd

print(check_even_or_odd(4))  # Output: Even
print(check_even_or_odd(5))  # Output: Odd
```

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

In Python, the concepts of iterators and iterables are closely related, but they serve different purposes.

(i) Iterable :

An iterable is any Python object capable of returning its members one at a time, allowing it to be iterated over in a for loop or with other iteration tools like map() and filter().
An iterable is any object that can provide an iterator. Examples of iterables include lists, tuples, strings, dictionaries, and sets.

How to check if an object is iterable: You can check if an object is iterable by using the iter() function. If iter() works without raising an error, the object is iterable.

Example -

```
my_list = [1, 2, 3]
if hasattr(my_list, '__iter__'):  # This checks if the object is iterable
    print("my_list is iterable")
```

Characteristics of Iterables:
Can be any collection like lists, tuples, strings, sets, dictionaries, etc.
They implement the __iter__() method, which returns an iterator.
They can be iterated over using loops or next().

Example -

```
my_list = [1, 2, 3]
# This is an iterable (a list), we can use it in a for loop.
for item in my_list:
    print(item)
```

(ii) Iterator :

An iterator is an object that represents a stream of data. It produces the next item in the sequence when asked, one item at a time, and knows how to keep track of its position.
An iterator must implement two methods -
(a) __iter__() – This method returns the iterator object itself.
(b) __next__() – This method returns the next item in the sequence. If there are no more items, it raises a StopIteration exception.

Characteristics of Iterators:
It keeps track of the current state or position while iterating over the sequence.
The __next__() method is used to get the next value.
Once the iterator has gone through all elements, calling __next__() raises a StopIteration exception to signal the end.

Creating an Iterator: An iterator can be created by calling iter() on an iterable (e.g., a list, tuple, etc.).

```
my_list = [1, 2, 3]
iterator = iter(my_list)  # Creates an iterator from the iterable

print(next(iterator))  # Output: 1
print(next(iterator))  # Output: 2
print(next(iterator))  # Output: 3
print(next(iterator))  # Raises StopIteration because the iterator is exhausted
```

Example -

```
# Example of a custom iterator
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

counter = Counter(1, 3)
for num in counter:
    print(num)  # Output: 1 2 3
```

(iii) Iterable vs Iterator in Action :

Example - Iterable (List) and Iterator

```
my_list = [10, 20, 30]
iterator = iter(my_list)  # Convert the list (iterable) to an iterator

print(next(iterator))  # Output: 10
print(next(iterator))  # Output: 20
print(next(iterator))  # Output: 30
# Next line will raise StopIteration since the iterator is exhausted
# print(next(iterator))  # StopIteration exception
```

The list is an iterable, which means you can pass it to iter() to get an iterator.
The iterator produced by iter(my_list) can then be used to retrieve elements one at a time with next().

> Iterable: An object that can be iterated over (e.g., lists, strings, tuples). It implements the __iter__() method, which returns an iterator.

> Iterator: An object that actually performs the iteration. It has a state (tracks the position during iteration) and implements both __iter__() and __next__() methods.

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

In Python, generators are a powerful and memory-efficient way to iterate over a sequence of values.
They allow you to define iterators in a simple and concise way, and they are especially useful when working with large datasets or infinite sequences because they produce items one at a time only when needed, without storing the entire sequence in memory.

Key Concepts of Generators :

(i) Generators are Iterators -
Like iterators, generators can be used in loops and support the __iter__() and __next__() methods.
However, generators are defined using a special syntax and do not need to implement the iterator protocol manually (like you would with custom iterators).

(ii) Generators are Lazy -

Generators produce values one at a time using a technique called lazy evaluation. They only generate the next value when requested (using the next() function or in a for loop).
This helps save memory, as the entire sequence does not need to be stored in memory at once.

(iii) yield keyword :
The defining characteristic of a generator is the yield keyword. The yield statement is used to produce a value and pause the function's execution, allowing it to be resumed when the next value is requested.
Each time the generator's __next__() method is called, execution continues from where it left off, using the last state of the function.

(iv) Generators vs Regular Functions :
Regular functions return a value and terminate, while generators use yield to return a value and can pause their execution, allowing them to yield more values over time without exiting.

Generators in Python are a way to create iterators in a memory-efficient manner.
They allow you to define sequences that are computed lazily, yielding values one by one as needed, instead of storing the entire sequence in memory.
The yield keyword is used to define a generator function, and the function can be used with next() or in a for loop.
Generators are ideal for working with large data sets or streams of data, offering significant memory and performance advantages.

Example -

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count  # Yielding value to the caller
        count += 1
# Using the generator
gen = count_up_to(5)

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
print(next(gen))  # Output: 4
print(next(gen))  # Output: 5
print(next(gen))  # Raises StopIteration, as the generator is exhausted
```

In this example, count_up_to(5) is a generator that produces values from 1 to 5. Each time you call next(gen), the generator yields the next value. Once the generator is exhausted (after reaching 5), calling next() raises a StopIteration exception.

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

Generators in Python offer several advantages over regular functions, especially when dealing with large datasets, streaming data, or situations where memory efficiency and performance are important.
Below are the key advantages of using generators:

(i) Memory Efficiency -

Generators are Lazy: Unlike regular functions that return all values at once, generators produce values one at a time and on-demand.
This means that the entire sequence of values doesn't need to be stored in memory, which can save a lot of memory, especially when working with large datasets or infinite sequences.

Example -
```
# With regular function
def generate_large_list(n):
    return [i for i in range(n)]  # Creates the entire list in memory

# With generator
def generate_large_numbers(n):
    for i in range(n):
        yield i  # Generates one value at a time, doesn't store the whole list in memory
```

In the second case, generate_large_numbers produces values one by one, so it doesn't need to hold the entire range in memory, making it more memory-efficient.

(ii) Improved Performance for Large Data -

Generators process data lazily, meaning they only compute the next value when requested, instead of calculating and storing the entire sequence in advance. This is particularly useful for handling large files, data streams, or long-running operations.
By processing data one element at a time, you avoid unnecessary computations and improve the performance of programs that need to work with large amounts of data.

Example: Reading large files line by line using a generator avoids reading the entire file into memory at once.

```
def read_large_file(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line  # Read one line at a time
```

(iii) Avoiding Unnecessary Computation :

Generators only compute values when needed. This means you don't perform unnecessary calculations or iterations. This can be useful when the full sequence might not even be used. For example, in cases where you only need the first few elements of a large data sequence, using a generator ensures you don't compute the rest of the data.

Example: If you only need the first 5 prime numbers, a generator will calculate primes one by one and stop when the required amount is reached, rather than computing all primes upfront.

(iv) Infinite Sequences :

Generators allow the creation of infinite sequences without consuming infinite memory. For example, you can create a generator for an infinite sequence like an ongoing count, Fibonacci numbers, or prime numbers. A regular function would need to store the entire sequence, which is impossible with infinite data.

Example: Infinite sequence of natural numbers -

```
def infinite_counter():
    count = 1
    while True:
        yield count  # Keeps generating numbers indefinitely
        count += 1

gen = infinite_counter()
for _ in range(5):
    print(next(gen))  # Output: 1, 2, 3, 4, 5
```

(v) Readable and Concise Code :

Generators simplify code: Instead of writing complicated logic to manage state and return multiple values over time, you can simply use yield inside a generator function to maintain state between iterations.

Example: A Fibonacci generator can be written more concisely using yield -

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

for num in fibonacci(5):
    print(num)  # Output: 0 1 1 2 3
```

This is much cleaner and easier to read than manually managing state with lists or other constructs.

(vi) State Preservation :

Generators preserve state between each call to next(). When a generator yields a value, it "pauses" and remembers its execution state (variables, execution point, etc.). When the generator is called again, it continues from where it left off. This makes them suitable for situations where the state needs to be maintained between iterations.

Example -
```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

gen = countdown(5)
print(next(gen))  # Output: 5
print(next(gen))  # Output: 4
```

The generator doesn't start from scratch on each call; it picks up where it left off.

(vii) Simplified Iteration :

Generators make it easy to iterate over sequences. Using a for loop on a generator automatically handles the StopIteration exception when the generator is exhausted. This eliminates the need to manually check whether more values are available or handle stopping conditions.

Example -
```
def square_numbers(n):
    for i in range(n):
        yield i ** 2

for num in square_numbers(5):
    print(num)  # Output: 0 1 4 9 16
```

With a regular function or approach, you might need additional logic to track whether you've processed all the elements, but generators simplify this.

(viii) Composing Generators :

Generators can be chained together, allowing you to compose multiple generator functions and create pipelines of operations. This allows for highly modular and efficient code.

Example: Chaining generators to first filter and then square the numbers -

```
def even_numbers(n):
    for i in range(n):
        if i % 2 == 0:
            yield i

def square_numbers(n):
    for i in range(n):
        yield i ** 2

# Chain the generators
for num in square_numbers(10):
    print(num)  # Output: 0 1 4 9 16 25 36 49 64 81
```

Generators provide a cleaner, more memory-efficient, and more performant way of working with sequences, especially in cases of large or infinite datasets.
By using the yield keyword, they offer lazy evaluation, meaning values are produced only when required, making them highly efficient and perfect for processing large amounts of data.

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

In Python, a lambda function is a small, anonymous function that can have any number of input parameters, but can only contain a single expression.
Lambda functions are defined using the lambda keyword instead of the standard def keyword used for defining regular functions.

Syntax -

```
    lambda arguments: expression
```

lambda : The keyword that indicates the creation of a lambda function.
arguments : A comma-separated list of input parameters (just like regular function arguments).
expression : A single expression that is evaluated and returned when the lambda function is called. It is typically a simple operation or computation.

Example -

```
# Lambda function to add two numbers
```

```
add = lambda x, y: x + y

print(add(5, 3))  # Output: 8
```

Here, lambda x, y: x + y defines a function that takes two arguments (x and y), and returns their sum (x + y).
The lambda function is assigned to the variable add, and then you can call it like a normal function.

Use Lambda Functions :

Lambda functions are often used in situations where you need a simple, one-off function that can be defined in a compact way. Here are some common use cases:

(i) Short Functions in Functional Programming -

Map, Filter, and Reduce functions often use lambda functions for short, on-the-fly operations.

Example - Using Lambda with map() :

```
# Using lambda to square each number in a list
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))

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

Example - Using Lambda with filter() :

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

print(even_numbers)  # Output: [2, 4, 6]
```

(ii) Sorting Lists -

Lambda functions are useful when sorting complex data structures like lists of tuples or dictionaries, where the sorting key needs to be determined dynamically.

Example - Sorting a List of Tuples :

```
# List of tuples (name, age)
```

```
people = [("Alice", 30), ("Bob", 25), ("Charlie", 35)]

# Sort by age (second element in each tuple)
people_sorted = sorted(people, key=lambda x: x[1])

print(people_sorted)  # Output: [('Bob', 25), ('Alice', 30), ('Charlie', 35)]
```

(iii) Used in Event Handlers or Callbacks -

Lambda functions are often used in places where functions are passed as arguments, such as event handlers, or callback functions in graphical user interfaces (GUIs) or asynchronous programming.

Example - Using Lambda in a Button Event Handler (GUI context) :

```
# Example for a GUI framework (e.g., Tkinter) where you need to define a simple action:
button = Button(root, text="Click Me", command=lambda: print("Button clicked"))
button.pack()
```

(iv) When a Simple Function is Needed Temporarily -

If you only need a function for a short period or for a small piece of code, it might be more convenient to use a lambda function rather than defining a full function with def.

Example - Lambda for Conditional Logic :

```
# Lambda to check if a number is positive
is_positive = lambda x: x > 0

print(is_positive(5))  # Output: True
print(is_positive(-3)) # Output: False
```

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

The map() function in Python is a built-in function that allows you to apply a given function to each item in an iterable (like a list, tuple, or string) and return a map object (an iterator) that yields the results.
The map() function is often used when you want to apply a transformation or operation to each item in a collection without using an explicit loop.

Syntax of map() -

```
  map(function, iterable, ...)
```

> function: A function that will be applied to each item of the iterable. This can be a regular function, a lambda function, or any callable.
> iterable: An iterable (such as a list, tuple, or string) whose items will be processed by the function.
> You can pass multiple iterables to the map() function, in which case the function must accept as many arguments as there are iterables.

The map() function returns an iterator (a map object), which you can convert into other data structures like a list or a tuple, if needed.

Example - Applying a function to each item in a list :

```
# A simple function to square a number
def square(x):
    return x ** 2

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Using map() to apply the square function to each number
squared_numbers = map(square, numbers)

# Converting the map object to a list and printing it
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

> In above example, the square function is applied to each element in the numbers list.
> map() returns a map object, which we convert to a list using list(), displaying the squared values.

Benefits of Using map() :-

> Compact and Concise: map() provides a more compact way to apply a function to all items in an iterable compared to using a for loop.
> Functional Programming: It is a core concept in functional programming and is ideal when you need to apply a transformation to a sequence of values.
> Performance: map() can be more efficient than using a for loop for large datasets, as it is implemented in C, which can speed up the operation.
> Clean Code: Using map() often results in cleaner and more readable code, especially when you need to perform simple transformations on a list or other iterable.

When to Use map() :

When you need to apply a single function to every item in an iterable.
When you want to avoid using an explicit for loop and want to use a more functional approach.
When you have a transformation or operation that can be neatly expressed in a single function (or lambda) and applied to a collection of data.

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

In Python, map(), reduce(), and filter() are higher-order functions that are often used in functional programming.
They all allow you to apply a function to elements of iterables, but they work in different ways and are used for different purposes.

(i) map() Function :

The map() function is used to apply a given function to each item of an iterable (like a list or tuple) and returns a map object (an iterator) with the results.

Syntax -

```
    map(function, iterable, ...)
```

function: A function that takes one or more arguments and returns a value.
iterable: An iterable (such as a list, tuple, or string) whose items will be processed by the function.

If multiple iterables are passed, the function should accept the same number of arguments as there are iterables.

Features of map() -

Applies a function to each item in an iterable and returns the results.
Can be used with one or more iterables.
It always returns an iterator (which can be converted to a list or other data structures).

Example -

```
# Example of map()
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x ** 2, numbers)

# Converting to a list to display the result
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

(ii) reduce Function :

The reduce() function is a part of the functools module and is used to apply a binary function (a function that takes two arguments) cumulatively to the items of an iterable, reducing the iterable to a single result.
It processes the items in the iterable from left to right (or right to left, depending on the function).

Syntax -

```
    from functools import reduce

    reduce(function, iterable, [initializer])
```

function: A binary function (i.e., a function that takes two arguments) that will be applied cumulatively.
iterable: An iterable whose items will be reduced.
initializer (optional): An initial value that will be used to start the reduction. If not provided, the first element of the iterable is used.

Features of reduce() -

Reduces an iterable to a single result by applying a function cumulatively.
Requires a binary function (a function that takes two arguments).
If the iterable has more than one item, the function is applied repeatedly across the elements, reducing them into one result.

Example -

```
from functools import reduce

# Example of reduce()
numbers = [1, 2, 3, 4, 5]

# Using reduce to calculate the product of all numbers
product = reduce(lambda x, y: x * y, numbers)

print(product)  # Output: 120
```

In this example, reduce() multiplies the numbers cumulatively: ((1 * 2) * 3) * 4 * 5 = 120.

(iii) filter() Function :

The filter() function is used to filter out elements from an iterable based on a condition provided by a function. It returns a filter object, which can be converted into a list or other iterable, containing only the elements that satisfy the condition.

Syntax -

```
    filter(function, iterable)
```

function: A function that defines the condition for filtering. The function should return True or False for each element of the iterable.
iterable: The iterable to be filtered.

The function should return True for elements you want to keep and False for elements you want to exclude. If the function is None, all elements that evaluate to True are kept.

Features of filter() -

Filters elements from an iterable based on a condition (whether the function returns True or False).
Returns a filter object (an iterator), which can be converted into other data types like a list.
The function passed should return a Boolean value (True or False).

Example -

```
# Example of filter()
numbers = [1, 2, 3, 4, 5, 6]

# Using filter to get only even numbers
even_numbers = filter(lambda x: x % 2 == 0, numbers)

# Converting the filter object to a list
print(list(even_numbers))  # Output: [2, 4, 6]
```

> map(): Transforms all elements of an iterable by applying a function to each element.
> reduce(): Reduces an iterable to a single value by applying a binary function cumulatively to the elements.
> filter(): Filters elements from an iterable based on a condition defined in a function, keeping only the elements where the function returns True.