# Assignment November 20,2024

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

Sol. A **function** is a piece of code that is called by name. It can be passed data to operate on (by the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed.

A **method** is a piece of code that is called by name that is associated with an object. In most respects, it is identical to a function except for two key differences:

1. A method is implicitly passed data to operate on by the object on which it was called.
2. A method is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data).

**Example:** A method is on an object or is static in class.

A function is independent of any object (and outside of any class).

For Java and C #, there are only methods. For C, there are only functions.

For C++ and Python it would depend on whether or not you're in a class. But in basic English:

* + **Function**: Standalone feature or functionality.
  + **Method**: One way of doing something, which has different approaches or methods, but related to the same aspect.

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

**Sol. Parameters:**

A parameter is the variable defined within the parentheses during function definition. Simply they are written when we declare a function.

**Example: Python**

# Here a,b are the parameters def sum(a,b):

print(a+b)

sum(1,2)

OUTPUT:

3

**Arguments:**

An argument is a value that is passed to a function when it is called. It might be a variable, value or object passed to a function or method as input. They are written when we are calling the function.

## Example: python

def sum(a,b):

print(a+b)

# Here the values 1,2 are arguments sum(1,2)

Output:

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

3

Sol. **How to Define a Function**

Defining a function in Python involves two main steps: defining the function and specifying the arguments it takes.

To define a function, you use the def keyword followed by the name of the function and parentheses (). If the function takes any arguments, they are included within the parentheses. The code block for the function is then indented after the colon.

Here's an example:

**def greet(name):**

**print("Hello, " + name + "! How are you?")**

In this example, we define a function called greet that takes one argument called name. The function then prints out a greeting message to the console that includes the name argument.

## How to Call a Function

Once you have defined a function, you can call it in your code as many times as you need.

To call a function in Python, you simply type the name of the function followed by parentheses (). If the function takes any arguments, they are included within the parentheses.

Here's an example:

greet("John")

In this example, we call the greet function with the argument "John". The output to the console would be:

Hello, John! How are you?

## Python Function Code Examples

Here's a complete code example that defines and calls the greet function:

def greet(name):

print("Hello, " + name + "! How are you?") greet("John")

When you run this code, it will output the following to the console:

Hello, John! How are you?

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

**Sol.** The return statement in Python is used to exit a function and pass a value back to the caller. It allows the function to output data that can be used elsewhere in the program. Without a return statement, a function will return None by default.

## Purpose of the return Statement:

1. **Exiting the function**: When the return statement is encountered, the function execution is stopped, and the program control is returned to the calling code.
2. **Returning values**: The return statement allows you to send a result from the function to the caller, enabling the use of that result for further processing.

## Example:

def add numbers(a, b):

result = a + b # Adding the two input numbers return result # Returning the result to the caller

# Calling the function and storing the returned value in a variable Sum result = add numbers (5, 3)

# Printing the result

print("The sum is:", sum result)

**Explanation:**

1. The function add numbers takes two parameters (a and b).
2. Inside the function, it calculates their sum and stores it in the variable result.
3. The return statement then returns this sum to the place where the function was called.
4. The calling code receives this value and stores it in sum result, which is then printed.

## Output:

The sum is: 8

In this case, the function uses the return statement to send the calculated sum back to the calling code, where it can be printed or used in other operations.

## What are iterators in Python and how do they differ from iterables? Sol. Iterators in Python

An **iterator** is an object in Python that allows you to traverse through all the elements of a collection, such as a list, tuple, or dictionary, one element at a time. An iterator implements two methods:

1. **iter ()**: This method returns the iterator object itself. It is used to initialize the iterator.
2. **next ()**: This method returns the next item from the collection. If there are no more items left to return, it raises the Stop Iteration exception to signal that the iteration is complete.

## Iterables in Python

An **iterable** is any Python object that can be looped over (iterated over). It is any object that can return an iterator using the iter() function. Examples of iterables include lists, tuples, strings, and dictionaries. An iterable object must implement the iter () method, which allows the use of iter() to obtain an iterator.

## Key Difference between Iterables and Iterators

* + **Iterable**: An iterable is an object that can return an iterator. It defines the collection, but it does not inherently have the ability to iterate through it. You need to call iter() on an iterable to get an iterator.
  + **Iterator**: An iterator is an object that is responsible for actually performing the iteration over the elements. It provides the next () method that fetches the next item in the collection.

## Example to Illustrate the Difference

# Example of an iterable

numbers = [1, 2, 3, 4] # This is an iterable (a list) # Calling iter() on the iterable to get an iterator

iterator = iter(numbers) # The iterator object is created

# Using the iterator to iterate over the elements print(next(iterator)) # Output: 1 print(next(iterator)) # Output: 2 print(next(iterator)) # Output: 3 print(next(iterator)) # Output: 4

# If we call next() again, it will raise StopIteration try:

print(next(iterator)) # This will raise StopIteration except StopIteration:

print("End of iteration")

**Explanation:**

1. **Iterable (numbers)**: The list numbers is an iterable because it can be used with the iter() function to get an iterator.
2. **Iterator (iterator)**: After calling iter(numbers), we get an iterator object. This object is responsible for the actual iteration.
3. **Iteration**: We use the next() function to get the next item from the iterator. After all elements are exhausted, the StopIteration exception is raised.

## Summary of Differences

|  |  |  |
| --- | --- | --- |
| **Feature** | **Iterable** | **Iterator** |
| Definition | Any object that can be iterated over (e.g., list, tuple). | An object that keeps track of the current position during iteration. |
| Methods | Implements iter () method. | Implements both iter () and  next () methods. |
| Creation | Can be passed directly to a loop or passed to iter() to create an  iterator. | Created by calling iter() on an iterable. |
| Reusability | Can be iterated multiple times by  creating new iterators. | Can only be traversed once. After it is  exhausted, it cannot be reset. |

**Example of Iterable with a For Loop:**

You can directly use an iterable in a for loop without manually calling iter() or next().

# Iterable example using for loop for number in numbers:

print(number)

This implicitly calls the iter() function to get the iterator, and then uses next() to iterate over the elements.

## Output:

1

2

3

4

In summary, an **iterable** provides a collection that can be looped through, while an

**iterator** is the object that allows the actual iteration process.

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

Sol.

In Python, **generators** are a special type of iterable, like lists or tuples, but they generate values on the fly, one at a time, as you iterate over them. This makes them more memory-efficient because they don't store the entire collection in memory.

A **generator** is a function that uses the yield keyword to return values one at a time. Each time the generator's next () method is called, it resumes execution from where it left off, keeping its local state.

## Example of a Generator

Here’s an example of a simple generator function:

def count\_up\_to(max): count = 1

while count <= max:

yield count # Yield the current count and suspend execution count += 1 # Increment count for the next call

# Create a generator counter = count\_up\_to(5)

# Using the generator with next() print(next(counter)) # Output: 1 print(next(counter)) # Output: 2 print(next(counter)) # Output: 3 print(next(counter)) # Output: 4 print(next(counter)) # Output: 5

# Calling next() again will raise StopIteration try:

print(next(counter)) # This will raise StopIteration except StopIteration:

print("End of iteration")

## Explanation:

1. **count\_up\_to(max)**: This is a generator function that yields numbers from 1 to max. Each time yield is encountered, it returns the current value of count and pauses the function’s execution.
2. **next(counter)**: The next() function is used to get the next value from the generator. Each call to next() continues from where the last yield was encountered.
3. Once the generator has yielded all values, further calls to next() will raise the StopIteration exception.

## Using Generators with Loops

You can also use a generator in a for loop. The loop automatically handles the iteration and stops when the generator is exhausted.

# Using the generator in a for loop for num in count\_up\_to(3):

print(num)

## Output:

1

2

3

**Conclusion:**

* + **Generators** in Python are a powerful and efficient way to handle large amounts of data lazily, one value at a time.
  + They are defined using functions with the yield keyword, allowing values to be produced on demand.
  + Compared to traditional iterables, generators are memory-efficient and suitable for dealing with large data sets or infinite sequences.

## What are the advantages of using generators over regular functions?

Sol. **Advantages of Using Generators Over Regular Functions**

Generators offer several advantages over regular functions, especially when working with large datasets or sequences where you don’t need to store all the values in memory at once. Below are the key advantages, followed by an example that compares the two approaches.

## Key Advantages of Generators

1. **Memory Efficiency**:
   * **Generators** are more memory-efficient than regular functions because they generate values one at a time and do not store the entire sequence in memory.
   * **Regular functions** that return large collections (like lists) need to store all values in memory, which can quickly become inefficient, especially when working with large datasets or infinite sequences.

## Lazy Evaluation:

* + **Generators** use lazy evaluation, meaning they only compute the next value when requested, which can save processing time when you don’t need the entire dataset immediately.
  + **Regular functions** compute all values at once and return them, even if you only need a small portion of them.

## Improved Performance:

* + **Generators** can lead to performance improvements when processing large datasets because they don’t require all the data to be loaded into memory at once.
  + **Regular functions** that return full collections may cause delays as they must first process all elements and then return them, which can be slow for large data.

## Cleaner and More Concise Code:

* + **Generators** allow you to create iterable sequences in a simple and readable way using yield, which can simplify the code compared to alternatives such as manually creating iterators or managing the state of an iteration.
  + **Regular functions** often require more complex code, such as building and managing lists or iterators.

## Handling Infinite Sequences:

* + **Generators** can handle infinite sequences since they produce values on demand without ever having to store them all in memory. For

example, you can generate an infinite sequence of numbers using a generator.

* + **Regular functions** would run out of memory or become infeasible when attempting to generate infinite sequences.

## Example: Comparing Regular Function and Generator Example 1: Using a Regular Function to Generate Numbers

Let's start by defining a regular function that returns a list of numbers from 1 to n:

def generate\_numbers(n): numbers = []

for i in range(1, n + 1): numbers.append(i)

return numbers

# Using the regular function result = generate\_numbers(5)

print(result) # Output: [1, 2, 3, 4, 5]

## Example 2: Using a Generator to Generate Numbers

Now, let's define a generator function that yields numbers from 1 to n:

def generate\_numbers\_gen(n): for i in range(1, n + 1):

yield i

# Using the generator

gen\_result = generate\_numbers\_gen(5) for num in gen\_result:

print(num)

## Output:

1

2

3

4

5

**Advantage 1: Memory Efficiency**

Consider the case where you need to generate a large sequence of numbers. Let’s compare the memory usage between using a regular function and a generator function.

# Regular Function

def generate\_large\_sequence(n):

numbers = [i for i in range(n)] # This creates a large list in memory return numbers

# Generator Function

def generate\_large\_sequence\_gen(n):

for i in range(n): # This yields numbers one at a time, using much less memory yield i

# Create a large sequence with 1 million numbers

large\_sequence = generate\_large\_sequence(10\*\*6) # Regular function stores all values

large\_sequence\_gen = generate\_large\_sequence\_gen(10\*\*6) # Generator yields values lazily

* **Regular function** stores all 1 million numbers in memory at once, which can cause memory consumption issues.
* **Generator function** yields each number one by one, which means it doesn’t hold the entire sequence in memory, making it more memory-efficient.

## Advantage 2: Lazy Evaluation

Consider a case where you only need to process the first few values of a sequence:

def process\_data():

for i in range(1, 1000000):

yield i # Imagine processing each number

gen = process\_data()

for \_ in range(10): # Only need the first 10 numbers print(next(gen))

With **lazy evaluation**:

* The generator only computes the first 10 numbers and doesn’t compute the rest, saving time and resources.
* A regular function would compute all numbers, even though you only need the first 10.

## Advantage 3: Handling Infinite Sequences

A regular function cannot generate infinite sequences because it tries to return all values at once, consuming all available memory. However, a generator can handle infinite sequences.

def infinite\_numbers(): num = 1

while True: # Infinite sequence yield num

num += 1

# Using the generator to get the first 10 numbers gen\_infinite = infinite\_numbers()

for \_ in range(10): print(next(gen\_infinite))

The **infinite sequence generator** will keep generating numbers indefinitely, without consuming unbounded memory.

## Conclusion

Generators in Python provide several advantages over regular functions, particularly when dealing with large or infinite sequences. Their memory

efficiency, lazy evaluation, and ability to handle infinite sequences make them a powerful tool for efficient data processing. Regular functions, on the other hand, can be simpler to use when working with small datasets where memory usage is not a concern.

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

**Sol. A** lambda **function** in Python is an anonymous (unnamed) function defined using the lambda keyword. Lambda functions can have any number of input parameters but only one expression. They are used when you need a simple function for a short period and do not want to define a full function using the def keyword**.**

## Syntax of a Lambda Function

lambda arguments: expression

* + **arguments**: The input parameters (can be multiple, separated by commas).
  + **expression**: A single expression that is evaluated and returned. Lambda functions do not have statements or multiple expressions.

## When to Use Lambda Functions?

Lambda functions are typically used in situations where:

1. A small, simple function is needed for a short period.
2. The function is used as an argument to higher-order functions (like map(), filter(), sorted(), etc.).
3. A full function definition is unnecessary and a more concise form is preferred.

## Example of a Lambda Function

1. ***Basic Example:***

Here’s a simple example of a lambda function that adds two numbers:

add = lambda x, y: x + y

# Using the lambda function result = add(5, 3) print(result) # Output: 8

In this example:

* + lambda x, y: x + y is a lambda function that takes two arguments (x and y) and returns their sum.
  + This is equivalent to defining a function using the def keyword like this:

def add(x, y): return x + y

## Using Lambda Functions with map(), filter(), and sorted()

Lambda functions are often used as arguments in functions like map(), filter(), and sorted() where a simple function is required.

* + **map()** applies a function to all items in an input list.

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

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

Here, the lambda function squares each number in the list.

* + **filter()** filters elements in a list based on a condition.

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers)) print(even\_numbers) # Output: [2, 4, 6]

In this case, the lambda function filters out only the even numbers.

* + **sorted()** sorts elements in a list based on a key.

pairs = [(1, 2), (3, 1), (5, 0), (4, 3)]

sorted\_pairs = sorted(pairs, key=lambda pair: pair[1]) print(sorted\_pairs) # Output: [(5, 0), (3, 1), (1, 2), (4, 3)]

Here, the lambda function sorts the list of tuples based on the second element of each tuple.

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

Sol. **Purpose of the map() Function in Python**

The **map()** function in Python is used to apply a given function to all items in an input iterable (such as a list, tuple, or string) and return a new iterable (usually a map object) with the results. The function you provide to map() is applied to each element of the iterable individually.

## Syntax of map()

map(function, iterable, ...)

* + **function**: The function that will be applied to each item of the iterable. This can be a regular function or a lambda function.
  + **iterable**: The iterable (like a list, tuple, etc.) whose elements the function will be applied to.
  + **... (optional)**: You can pass multiple iterables to map() if the function accepts multiple arguments. The function will be applied to the elements of all iterables in parallel.

The map() function returns an **iterator** that produces the results one by one as you loop over it.

## Common Use Cases of map()

* + Transforming or modifying each item of an iterable.
  + Applying a function to multiple iterables simultaneously (if the function accepts multiple arguments).
  + In combination with lambda functions for more concise code.

## Example 1: Using map() with a Regular Function

Let’s say you have a list of numbers and you want to square each number. You can define a function and use map() to apply it to the list.

# Regular function to square a number def square(x):

return x \*\* 2 numbers = [1, 2, 3, 4, 5]

# Apply the 'square' function to each element in the list using map() squared\_numbers = map(square, numbers)

# Convert the result to a list and print print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

## Explanation:

* + square(x) is a function that squares a number.
  + map(square, numbers) applies the square function to each item in the numbers list.
  + The map() function returns an iterator, so we convert it to a list using list() to see the results.

## Example 2: Using map() with a Lambda Function

You can use a **lambda function** to make the code more concise. Here's the same example using a lambda instead of a regular function:

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

# Use map with a lambda function to square each element squared\_numbers = map(lambda x: x \*\* 2, numbers)

# Convert to list and print

print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

## Example 3: Using map() with Multiple Iterables

If you have multiple iterables and want to apply a function that takes multiple arguments, you can pass several iterables to map().

# Two lists of numbers numbers1 = [1, 2, 3]

numbers2 = [4, 5, 6]

# Function to add corresponding elements def add(x, y):

return x + y

# Apply the 'add' function to pairs of elements from both lists result = map(add, numbers1, numbers2)

# Convert to list and print print(list(result)) # Output: [5, 7, 9]

## Explanation:

* + The function add(x, y) takes two arguments and returns their sum.
  + map(add, numbers1, numbers2) applies the add() function to each pair of elements from numbers1 and numbers2, returning a new iterable with the results.
  + The output is [5, 7, 9], where each element is the sum of the corresponding elements from the two lists.

## When to Use map():

* + When you need to transform all elements of an iterable with the same function.
  + When you want to avoid using explicit loops.
  + When working with large datasets where memory efficiency (via lazy evaluation) is important.

## Conclusion

The map() function is a powerful tool for applying a function to all elements of an iterable, returning an iterator with the results. It's commonly used for transforming data and works well with both regular functions and lambda functions. It can also handle multiple iterables simultaneously, making it a versatile and concise way to process collections.

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

**Sol.** In Python, the map(), reduce(), and filter() functions are all part of the functional programming paradigm, and they are used to operate on iterables (like lists, tuples, etc.). Although they may seem similar, each function serves a distinct purpose.

## map(): Applies a Function to All Items in an Iterable

* + **Purpose**: The map() function applies a given function to all items in an iterable (e.g., a list, tuple, etc.) and returns a new iterable (typically a map object) with the results.
  + **Usage**: It is used when you want to transform or modify each element in an iterable.
  + **Result**: The output is an iterator, which can be converted to a list, tuple, etc.

## Example of map():

# Function to square a number def square(x):

return x \*\* 2

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

# Use map to square each number squared\_numbers = map(square, numbers)

# Convert the map object to a list and print print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

Here, map() applies the square function to each element in the numbers list.

## reduce(): Applies a Function to Accumulate Results Across an Iterable

* + **Purpose**: The reduce() function from the functools module applies a binary function (a function that takes two arguments) cumulatively to the items in an iterable, from left to right, so as to reduce the iterable to a single value.
  + **Usage**: It is used when you want to accumulate or combine elements from an iterable into a single value.
  + **Result**: The output is a single accumulated value.

## Example of reduce():

from functools import reduce

# Function to add two numbers def add(x, y):

return x + y

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

# Use reduce to sum all numbers result = reduce(add, numbers)

print(result) # Output: 15

In this example, reduce() starts with the first two elements, adds them, then takes the result and adds it to the next element, and so on until all elements have been processed. The final result is the sum of all numbers: 15.

## filter(): Filters Items in an Iterable Based on a Condition

* + **Purpose**: The filter() function filters the elements of an iterable based on a condition defined by a function. Only elements for which the function returns True are kept in the result.
  + **Usage**: It is used when you want to filter out elements that don't meet a certain condition.
  + **Result**: The output is an iterator containing only the elements for which the function returned True.

## Example of filter():

# Function to check if a number is even def is\_even(x):

return x % 2 == 0

# List of numbers numbers = [1, 2, 3, 4, 5, 6]

# Use filter to get only even numbers even\_numbers = filter(is\_even, numbers)

# Convert the filter object to a list and print print(list(even\_numbers)) # Output: [2, 4, 6]

In this example, filter() keeps only the numbers for which the is\_even() function returns True.

## Example Comparing All Three Functions

Let’s look at an example where we apply all three functions to the same dataset.

## Example: Using map(), reduce(), and filter() Together

from functools import reduce

# Sample list of numbers

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 1. Use map() to square each number squared\_numbers = map(lambda x: x \*\* 2, numbers)

print("Squared Numbers:", list(squared\_numbers)) # Output: [1, 4, 9, 16, 25, 36, 49,

64, 81]

# 2. Use filter() to get only even numbers even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print("Even Numbers:", list(even\_numbers)) # Output: [2, 4, 6, 8]

# 3. Use reduce() to get the sum of all numbers total\_sum = reduce(lambda x, y: x + y, numbers) print("Sum of Numbers:", total\_sum) # Output: 45

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

**Sol.** To explain the internal mechanism of the reduce() function for summing the values in a given list [47, 11, 42, 13], let's break down the process step by step.

Step-by-Step Mechanism of reduce()

The reduce() function from the functools module takes two arguments:

1. A function that takes two inputs.
2. An iterable (in this case, the list [47, 11, 42, 13]).

The function applies the binary function (like addition) cumulatively to the elements of the iterable, from left to right, reducing the iterable to a single value.

## Example:

from functools import reduce

# List of numbers numbers = [47, 11, 42, 13]

# Using reduce to sum the numbers

result = reduce(lambda x, y: x + y, numbers) print(result)

Here, the binary function is lambda x, y: x + y, which simply adds two numbers. The operation happens as follows:

Internal Steps for Summing the List [47, 11, 42, 13] using reduce():

1. Initial Step:
   * The first two elements of the list are taken: 47 and 11.
   * x = 47, y = 11
   * The lambda function is called: 47 + 11 = 58.
   * Intermediary result: The result after this step is 58.
2. Second Step:
   * Now, the result from the previous step (58) is used with the next element in the list (42).
   * x = 58, y = 42
   * The lambda function is called: 58 + 42 = 100.
   * Intermediary result: The result after this step is 100.
3. Third Step:
   * The result from the previous step (100) is used with the next element in the list (13).
   * x = 100, y = 13
   * The lambda function is called: 100 + 13 = 113.
   * Final result: The result after this step is 113. Summary of the reduce() Function's Execution:
4. Start with 47 and 11: 47 + 11 = 58
5. Use 58 and 42: 58 + 42 = 100
6. Use 100 and 13: 100 + 13 = 113

Therefore, the final result of applying reduce() with the summing function on the list [47, 11, 42, 13] is 113.

Final Result:

113

Visual Representation:

Here's how it looks on paper in terms of internal operations:

Step 1:

(47 + 11) = 58

Step 2:

(58 + 42) = 100

Step 3:

(100 + 13) = 113

Final result = 113

# Practical Questions:

1. **Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list**.

**Python Function:**

def sum\_of\_even\_numbers(numbers): # Initialize the sum to 0

total = 0

# Loop through each number in the list for num in numbers:

# Check if the number is even if num % 2 == 0:

total += num # Add the even number to the total

# Return the total sum of even numbers return total

**Example Usage:**

numbers = [10, 15, 22, 33, 44, 55, 60]

result = sum\_of\_even\_numbers(numbers) print(result) # Output: 136 (10 + 22 + 44 + 60)

1. **Create a Python function that accepts a string and returns the reverse of that string.**

To create a Python function that accepts a string and returns the reverse of that string, we can use Python's string slicing feature, which allows us to easily reverse a string.

## Function Definition:

def reverse\_string(input\_string):

# Use slicing to reverse the string return input\_string[::-1]

**Explanation:**

* + **[::-1]**: This is a Python slicing technique that creates a new string starting from the end to the beginning, effectively reversing the original string.

## Example Usage:

# Test the function original\_string = "Hello, World!"

reversed\_string = reverse\_string(original\_string) print(reversed\_string) # Output: "!dlroW ,olleH"

**Explanation:**

* + The function reverse\_string() takes the input string "Hello, World!" and reverses it using string slicing.
  + The result is "!dlroW ,olleH".

## Edge Case:

* + If an empty string is passed, the function will return an empty string as well. empty\_string = ""

reversed\_empty = reverse\_string(empty\_string)

print(reversed\_empty) # Output: ""

This function works efficiently and concisely to reverse any string passed to it.

1. **Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.**

**Python Function:**

def square\_numbers(numbers):

# Use a list comprehension to square each number return [num \*\* 2 for num in numbers]

**Explanation:**

* + **List comprehension**: [num \*\* 2 for num in numbers] iterates through each element in the list numbers, squares each element (num \*\* 2), and collects the results in a new list.

**Example Usage:**

# Test the function numbers = [1, 2, 3, 4, 5]

squared\_list = square\_numbers(numbers) print(squared\_list) # Output: [1, 4, 9, 16, 25]

**Explanation:**

* The input list is [1, 2, 3, 4, 5].
* The function squares each element:

o 1^2 = 1

o 2^2 = 4

o 3^2 = 9

o 4^2 = 16

o 5^2 = 25

* The result is a new list [1, 4, 9, 16, 25].

**Edge Cases:**

1. **Empty List**: If the input list is empty, the function will return an empty list as well.

empty\_list = []

result = square\_numbers(empty\_list) print(result) # Output: []

1. **Negative Numbers**: The function will correctly square negative numbers as well.

numbers = [-1, -2, -3]

squared\_list = square\_numbers(numbers) print(squared\_list) # Output: [1, 4, 9]

1. **Write a Python function that checks if a given number is prime or not from 1 to 200.**

**Python Function:**

import math

def is\_prime(n):

# Check if the number is less than 2 (not prime) if n <= 1:

return False

# 2 is a prime number elif n == 2:

return True

# Eliminate even numbers greater than 2 elif n % 2 == 0:

return False

# Check for factors from 3 to sqrt(n), only odd numbers for i in range(3, int(math.sqrt(n)) + 1, 2):

if n % i == 0: return False

return True

**Example Usage:**

# Check prime numbers from 1 to 200 for num in range(1, 201):

if is\_prime(num): print(num)

**Expected Output:**

This will print all the prime numbers from 1 to 200, such as: 2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

61

67

71

73

79

83

89

97

101

103

107

109

113

127

131

137

139

149

151

157

163

167

173

179

181

191

193

197

199

## Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

Sol.

**class FibonacciIterator:**

def init (self, n):

# Initialize the iterator with the number of terms (n) self.n = n

self.count = 0

self.a, self.b = 0, 1 # Starting values for Fibonacci sequence def iter (self):

# Return the iterator object itself return self

def next (self):

# If we have already generated n terms, stop iteration if self.count >= self.n:

raise StopIteration

# Generate the next Fibonacci number current = self.a

self.a, self.b = self.b, self.a + self.b # Update to next Fibonacci numbers self.count += 1

return current

**Output:**

0

1

1

2

3

5

8

13

21

1. **Write a generator function in Python that yields the powers of 2 up to a given exponent.**

**Python Generator Function:**

def powers\_of\_2(n):

# Start from exponent 0 to n (inclusive) for i in range(n + 1):

yield 2 \*\* i # Yield the power of 2 for the current exponent

Output:

1

# 2^0

2

# 2^1

4

# 2^2

8

# 2^3

16 # 2^4

32 # 2^5

## Implement a generator function that reads a file line by line and yields each line as a string.

**Sol. Python Generator Function**:

def read\_file\_line\_by\_line(filename): # Open the file in read mode

with open(filename, 'r') as file: # Read the file line by line for line in file:

yield line.strip() # Use strip() to remove leading/trailing whitespace/newlines

Hello, World! This is a test file.

It contains multiple lines. Goodbye!

Output:

Hello, World! This is a test file.

It contains multiple lines. Goodbye!

## Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.

**Sol. Sorting with sorted():**

List of tuples

tuples\_list = [(1, 3), (2, 1), (4, 2), (3, 4)]

# Sort the list of tuples based on the second element using a lambda function sorted\_list = sorted(tuples\_list, key=lambda x: x[1])

# Print the sorted list print(sorted\_list)

**Output:**

[(2, 1), (4, 2), (1, 3), (3, 4)]

1. **Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit.**

**Sol. Python Code:**

# List of temperatures in Celsius celsius\_temperatures = [0, 20, 30, 40, 100]

# Define a lambda function to convert Celsius to Fahrenheit fahrenheit\_temperatures = map(lambda c: (9/5) \* c + 32, celsius\_temperatures)

# Convert the map object to a list and print the result fahrenheit\_temperatures\_list = list(fahrenheit\_temperatures) print(fahrenheit\_temperatures\_list)

**Output:**

**[32.0, 68.0, 86.0, 104.0, 212.0]**

**11.Create a Python program that uses `filter()` to remove all the vowels from a given string.**

**Sol.**

# Function to remove vowels from a string using filter def remove\_vowels(input\_string):

# Define vowels

vowels = "aeiouAEIOU"

# Use filter to remove vowels and join the result into a string

filtered\_string = ''.join(filter(lambda char: char not in vowels, input\_string))

return filtered\_string

# Example usage

input\_string = "Hello, World!"

result = remove\_vowels(input\_string) print(result)

**Output:**

Hll, Wrld!

**11) Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:**

## Order no book title and author quantity price per item 34578 learning python 4 40.95

## 98762 programming python 5 56.80

## 77226 head first python 3 32.95

## 88112 einfuhrung in python 3 34.99

**Write a Python program, which returns a list with 2-tuples. Each tuple consists of the order number and the product of the price per item and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €. Write a Python program using lambda and map.**

orders = [ ["34587", "Learning Python, Mark Lutz", 4, 40.95],["98762", "Programming Python, Mark Lutz", 5, 56.80],

["77226", "Head First Python, Paul Barry", 3,32.95],

["88112", "Einführung in Python3, Bernd Klein", 3, 24.99]] invoice\_totals = list(map(lambda x: x if x[1] >= min\_order else (x[0], x[1] + 10),map(lambda x: (x[0],x[2] \* x[3]), orders)))

#Note- To understand the working of above lambda function break the function till innermost map function. Break and understand in below fashion

'''output1 = map(lambda x: (x[0],x[2] \* x[3]), orders) #Innermost lambda function execution

output2 = map(lambda x: x if x[1] >= min\_order else (x[0],

x[1] + 10),map(lambda x: (x[0],x[2] \* x[3]), orders) final = list(map(lambda x: x if

x[1] >= min\_order else (x[0], x[1] + 10),map(lambda x: (x[0],x[2] \* x[3]), orders)))

Output:

map(lambda y:y[1]\*y[2] ,x[1:])