Theory Questions:- Functions

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

- FUNCTION:
- Definition: A block of code that performs a task.
- Calling: called directly using its name.
- Association: Not associated with any object.
- Example Use: General purpose code reusable anywhere.
    - **function example**

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


print(greet("Riya"))  # Calling a function directly
                        # Output: Hello, Riya!
```

- METHOD:
- Definition: A function that **belongs to an object**.
- Calling: Called using the object (e.g., obj.method()).
- Association: Always associated with an object or class.
- Example use: Works specifically with the object it belongs to.
    - METHOD EXAMPLE:

```
# Method example
text = "hello world"
print(text.upper())    # Calling a method belonging to string object
                       # Output: HELLO WORLD
```

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

- Explanation: Parameters → Variables defined in the function definition that accept input values.
- Arguments → Actual values passed to the function when it is called.
- Python supports different types of arguments:

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Example:

```
# Function with parameters
def greet(name, message="Good Morning"):
return f"Hello {name}, {message}!"

# Positional argument
print(greet("Riya"))    # Output: Hello Riya, Good Morning!

# Keyword argument
print(greet(name="Riya", message="Welcome to Python"))   # Output: Hello Riya, Welcome to Python!

# Default argument
```

```
print(greet("Riya"))
# Output: Hello Riya, Good Morning!
```

Q3. Different ways to define and call a function in Python?

- Explanation: Functions are reusable blocks of code that perform a specific task. In Python, you can define and call functions in multiple ways:

1. Standard function (def) Defined using the def keyword. Can accept parameters and return values.
2. Lambda function Also called an anonymous function. Defined using lambda keyword, usually for simple, single-line tasks.
3. Nested function A function defined inside another function. Useful for encapsulation and keeping helper functions local to the outer function.

- Calling a function involves using its name followed by parentheses () .
- Arguments can be passed when calling functions to provide input values.

Examples:

```
# 1. Standard function
def greet(name):
return f"Hello {name}!"

print(greet("Riya"))  # Calling standard function # Output: Hello Riya!

# 2. Lambda function
square = lambda x: x**2
print(square(5))        # Calling lambda function  # Output: 25

# 3. Nested function
def outer():
def inner():
            return "This is a nested function"
```

```
    return inner()      # Calling inner function inside outer

print(outer())      # Output: This is a nested function
```

Double-click (or enter) to edit

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

- Explanation: The return statement is used to send a result from a function back to the caller.
- Without return , a function returns None by default. return allows functions to produce a value that can be stored, printed, or used in other operations.
- It also helps in breaking out of a function immediately.

Examples: # Function using return def greet(name): return f"Hello {name}!"

```
# Using the returned value
message = greet("Riya")
print(message)               # Output: Hello Riya!

# Function without return def greet_no_return(name):
print(f"Hello {name}!")   # Output: Hello Riya!

result = greet_no_return("Riya"):
print(result)                # Output: Will print None
```

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

- Explanation:
- Iterable: Any Python object capable of returning its elements one by one, e.g., list , tuple , string , set , or dictionary .
    - You can loop over it using a for loop.

- Iterator: An object that produces elements one at a time and keeps track of its current position.
    - Created from an iterable using the iter() function.
    - Use next() to access elements individually.
- Key Differences Between Iterable and Iterator:
    - Iratable:
- Definition: can be looped over to access elements.
- object type: any collection type (list,tuplke,etc)
- Access: use foe loop
- Exhaustible: No
    - Iterator:
- Definition: produces elements one by one on demand.
- Object type: Created from an iterable using iter()
- Access: Use next () Function
- Exhaustible: Yes,once traversed, cannot be reused.

Example:

```
# Iterable example
fruits = ["apple", "banana", "cherry"]  # This is an iterable

       print("Iterable output:")
       for fruit in fruits:
       print(fruit)                          # Output: apple

                                             # Output: banana
```

```
                                        # Output: cherry

# Iterator example
fruits_iter = iter(fruits)              # Convert iterable to iterator

print("\nIterator output using next():")

print(next(fruits_iter))       # Output: apple

print(next(fruits_iter))       # Output: banana

print(next(fruits_iter))       # Output: cherry
```

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

- Explanation: Generators are special iterators that generate values on the fly instead of storing them in memory.
- They are memory-efficient and useful for large datasets or streams of data.
- Generators are defined in two main ways:
    1. Using a function with the yield statement.
    2. Using generator expressions (similar to list comprehensions but with parentheses).
- Each call to next() on a generator produces the next value until the generator is exhausted.

- Example using yield

```
# Generator function
def greet_generator(names):
for name in names:
  yield f"Hello {name}!"

  names_list = ["Riya", "Alex", "Maya"]
```

```
# Create generator
 gen = greet_generator(names_list)


# Access values one by one
print(next(gen))   # Output: Hello Riya!
print(next(gen))   # Output: Hello Alex!
print(next(gen))   # Output: Hello Maya!
```

- Example using generator expression:

```
# Generator expression
squares = (x**2 for x in range(1, 4))

print(next(squares))  # Output: 1
print(next(squares))  # Output: 4
print(next(squares))  # Output: 9
```

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

- Explanation:

  Generators provide several advantages over regular functions that return lists:

1. Memory Efficiency
    - Generators produce values one at a time and do not store the entire sequence in memory.
    - Useful for large datasets.
2. Lazy Evaluation
    - Values are computed only when needed using next() .

- Reduces unnecessary computations.

3. Represent Infinite Sequences
   - Generators can model infinite series (like Fibonacci numbers) which is impossible with lists.
4. Pipeline Processing
   - Generators can be chained together to create pipelines, processing data efficiently.

Example:

```
# Generator function

def greet_generator(names):
for name in names:
yield f"Hello {name}!"

   names_list = ["Riya", "Alex", "Maya"]

gen = greet_generator(names_list)

print(next(gen))  # Output: Hello Riya!
print(next(gen))  # Output: Hello Alex!
print(next(gen))  # Output: Hello Maya!

# Memory efficiency demonstration
numbers = (x**2 for x in range(1, 1000000))  # Generator for 1 million squares
print(next(numbers))  # Output: 1
print(next(numbers))  # Output: 4
print(next(numbers))  # Output: 9
```

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

- Explanation:
- Lambda functions are anonymous, one-line functions in Python that do not require a formal def block.
- They are defined using the lambda keyword, followed by arguments and a single expression.
- Lambda functions are commonly used when you need a small, temporary function without cluttering your code.
- Typical use cases include functional programming, inline operations, and passing functions as arguments to higher-order functions like map() , filter() , or sorted() .
- Advantages: concise, readable for small operations, and memory-efficient since they don't require a full function definition.

Syntax:

```
lambda arguments: expression
```

Characteristics & Usage:

- Concise — Perfect for small, one-off operations.
- Inline — Can be defined and used in a single line.
- Functional programming — Ideal for passing as arguments to functions like map() , filter() , or reduce() .
- Single expression only — Cannot contain multiple statements or complex logic.
- Temporary & lightweight — Great for quick tasks without cluttering your code with full function definitions.

Example 1: Basic Lambda

```
# Lambda to greet Riya
greet = lambda name: f"Hello {name}!"
print(greet("Riya"))  # Output: Hello Riya!
```

Example 2: Lambda with map()

```
numbers = [1, 2, 3, 4, 5]
# Square each number using lambda
```

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

Example 3: Lambda with filter()

```
# Filter numbers greater than 2
filtered = list(filter(lambda x: x > 2, numbers))
print(filtered)  # Output: [3, 4, 5]
```

Example 4: Lambda with reduce()

```
from functools import reduce
# Sum all numbers using reduce and lambda
total = reduce(lambda x, y: x + y, numbers)
print(total)  # Output: 15
```

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

Explanation:

- The map() function applies a given function to each item of an iterable (like a list, tuple, etc.) and returns a map object, which is an iterator.
- Useful when you want to perform the same operation on multiple items without using a loop.
- Lazy evaluation: the values are computed only when iterated, saving memory for large datasets.

Common Uses of map() :

- Transforming all elements in a list/tuple without using a loop. - - - Applying mathematical operations to all elements.
- Preprocessing or cleaning data in bulk, e.g., converting strings to uppercase.
- Chaining with filter() or reduce() for functional programming pipelines.
- Syntax:

map(function, iterable)

Example 1: Basic map with a named function

```
# Function to greet a person
def greet(name):
return f"Hello {name}!"
names = ["Riya", "Alex", "Maya"]
# Apply greet function to each name
greetings = map(greet, names)
# Convert to list and print
print(list(greetings))
# Output: ['Hello Riya!', 'Hello Alex!', 'Hello Maya!']
```

Example 2: map with lambda function

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

Example 3: map with multiple iterables

```
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
# Add corresponding elements from two lists
sum_list = map(lambda x, y: x + y, numbers1, numbers2)
print(list(sum_list))
# Output: [5, 7, 9]
```

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

- Explanation:

Python provides built-in functional programming tools like map() , reduce() , and filter() to process iterables efficiently:

- Key Differences:

1. Purpose:
    - map() → changes/transforms every element in an iterable.
    - filter() → chooses only certain elements that satisfy a condition.
    - reduce() → combines all elements to produce a single result.
2. Return Type:
    - map() and filter() return iterators, which can be converted to a list.
    - reduce() returns a single value.
3. Number of Iterables:
    - map() can work with one or more iterables.
    - filter() works with one iterable.
    - reduce() works with one iterable.
4. Use Case Example:
    - map() → Transforming [1,2,3] into [1,4,9] (squares).
    - filter() → Picking [3,4,5] from [1,2,3,4,5] (numbers >2).
    - reduce() → Summing [1,2,3,4,5] into 15 .

Example 1: map()

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

# Square each number
```

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

Example 2: filter()

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

# Keep numbers greater than 2
filtered = filter(lambda x: x > 2, numbers)
print(list(filtered))     # Output: [3, 4, 5]
```

Example 3: reduce()

```
from functools import reduce

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

# Sum all numbers cumulatively
total = reduce(lambda x, y: x + y, numbers)
print(total)              # Output: 1
```

Q11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list: [47,11,42,13]; (Attach paper image for this answer) in doc or collab notebook.

Reduce function = Sum operation (Step by Step)

Given List : [47, 11, 42, 13]

Internal working of reduce (lambda x, y : x+y) [47, 11, 42, 43].

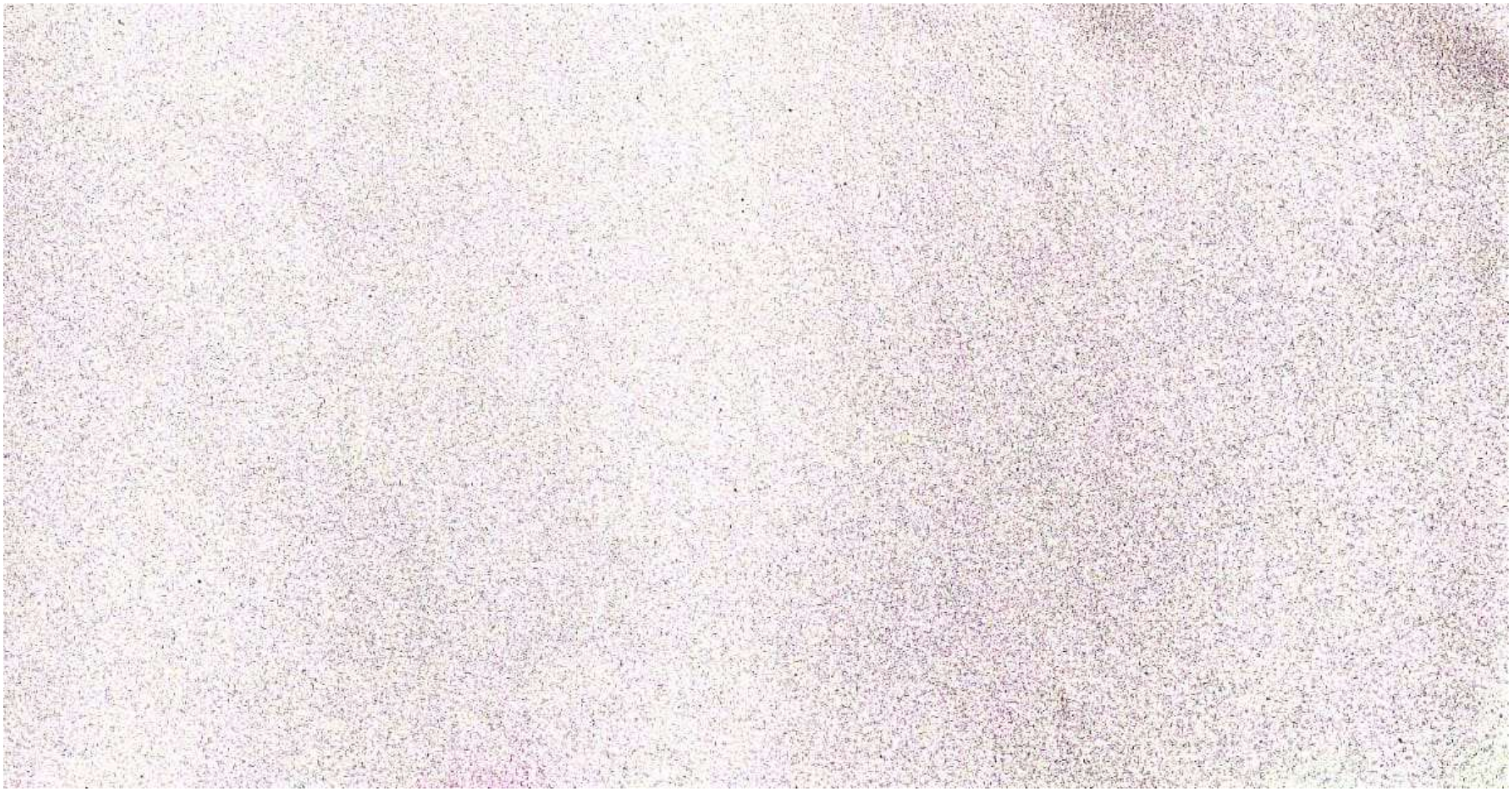
Step 1: Take first two Numbers → 47 + 11 = 58.

Step 2: Take result & next number → 58 + 42 = 100.

Step 3: Take Result & next number → 100 + 13 = 113

→ Final Result : 113.

- So, reduce processes pair by pair until one find Result Remains.

- 

:

→

Practical Questions:

Q1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.

- ```
  def sum_of_evens(nums):
  return sum(n for n in nums if n % 2 == 0)


  print(sum_of_evens([1, 2, 3, 4, 5, 6, 7, 8]))
  ```

Q2. Create a Python function that accepts a string and returns the reverse of that string.

- ```
    def reverse_string(s):
      return s[::-1]

    print(reverse_string("Riya"))
  ```

Q3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.

- ```
      def square_list(numbers):
       return [num ** 2 for num in numbers]

  print(square_list([1, 2, 3, 4, 5]))
  ```
  ```

```
[1, 4, 9, 16, 25]
```

Q4. Write a Python function that checks if a given number is prime or not from 1 to 200.

- ```
  def is_prime(num):
     if num < 2:
       return False
  for i in range(2, num):
   if num % i == 0:
       return False
  return True

    def prime_numbers_upto_200():
      primes = []
       for n in range(1, 201):
        if is_prime(n):
         primes.append(n)
     return primes


    print(prime_numbers_upto_200())

    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,]
  ```

Q5. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

- ```
  class FibonacciIterator:
   def __init__(self, n_terms):
  self.n_terms = n_terms   # total terms required
  ```

```
        self.count = 0           # counter
        self.a, self.b = 0, 1    # starting values

         def __iter__(self):
        return self

        def __next__(self):
        if self.count < self.n_terms:
            value = self.a
            self.a, self.b = self.b, self.a + self.b
            self.count += 1
            return value
        else:
            raise StopIteration

    fib = FibonacciIterator(10)     # generate first 10 terms
    for num in fib:

      print(num, end=" ")

      0 1 1 2 3 5 8 13 21 34
```

Q6. . Write a generator function in Python that yields the powers of 2 up to a given exponent.

- ```
  def powers_of_two(n):
   for i in range(n + 1):
   yield 2 ** i

   for num in powers_of_two(5):
   print(num, end=" ")
  ```
  ```

```
1 2 4 8 16 32
```

Double-click (or enter) to edit

Q7. Implement a generator function that reads a file line by line and yields each line as a string.

- ```
  # Step 1: Create a sample file
  with open("sample.txt", "w") as f:
  f.write("Hello\n")
  f.write("This is Riya\n")
  f.write("Learning Python\n")


  # Step 2: Generator function to read file line by line
  def read_file_line_by_line(filename):
  with open(filename, "r") as file:
   for line in file:
       yield line.strip()   # strip() removes \n at the end

  # Step 3: Use the generator
   for line in read_file_line_by_line("sample.txt"):
   print(line)


   Hello
   This is Riya
   Learning Python
  ```

Q8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple
  ```

- ```
  def sort_tuples_by_second(tuples_list):
  return sorted(tuples_list, key=lambda x: x[1])


  data = [(1, 5), (3, 1), (4, 7), (2, 3)]
  result = sort_tuples_by_second(data)
  print(result)


 [(3, 1), (2, 3), (1, 5), (4, 7)]
  ```

Double-click (or enter) to edit

Q9.Write a Python program that uses map() to convert a list of temperatures from Celsius to Fahrenheit.

- ```
  # Function to convert Celsius to Fahrenheit
  def celsius_to_fahrenheit(celsius):
  return (celsius * 9/5) + 32

  # List of temperatures in Celsius
  celsius_list = [0, 20, 37, 100]

  # Using map() to convert all temperatures
  fahrenheit_list = list(map(celsius_to_fahrenheit, celsius_list))


  print("Celsius:", celsius_list)
  print("Fahrenheit:", fahrenheit_list)
  ```

```
Celsius: [0, 20, 37, 100]
Fahrenheit: [32.0, 68.0, 98.6, 212.0]
```

Q10 Create a Python program that uses filter() to remove all the vowels from a given string

- ```
  # Function to check if a character is NOT a vowel
  def is_not_vowel(ch):
    vowels = "aeiouAEIOU"
   return ch not in vowels

   # Input string
   text = "Hello, This is Riya"

    # Using filter() to remove vowels
    result = ''.join(filter(is_not_vowel, text))

    print("Original String:", text)
    print("After Removing Vowels:", result)

     Original String: Hello, This is Riya
     After Removing Vowels: Hll, Hll, Ths s Ry
  ```

Q11. Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

```
- 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
```

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.

```
 # Accounting routine with lambda and map

orders = [
  [34587, "Learning Python, Mark Lutz", 4, 40.95],
```