Functions

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

=> Functions and methods in Python are both blocks of reusable code that perform specific tasks. But there are some differences between them;

Function	Method
Standalone block of code that can perform some specific computation and it can be reused.	Those functions are associated with a particular object or class.
It can be called from anywhere within the program.	Only be called on instances of that class.
Don’t belong to a specific object or class.	Provide functionality specific to the object or class they belong to.

Example:

```
# Function definition;

def greeting (name):

        print(“Welcome to the class”, name,”!”)


# Call the function

greeting(“Rio”)    #Output: “Welcome to the class Rio!”
```

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

=> In Python, function arguments and paraments are closely related concepts that refer to the values passed to a function when it’s called.

Parameters:

- Defined within the function’s definition.
- Used to receive values from the caller.
- Act as placeholders for the actual values that will be passed to the function.

Arguments:

- The actual values passed to a function when it’s called.

- Correspond to the parameters defined in the function’s definition.

Examples:

```
def greeting (name):
    print(“Welcome to the class”, name,”!”)


# Calling the function with an argument
greeting(“Rio”)    #Output: “Welcome to the class Rio!”
```

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

=>

Defining Functions:

There are two primary ways to define functions in Python;

Using the **def** keyword,

- This is the most common method.
- The function name, parameters (if any), and the function body are enclosed within parentheses and a colon.
- The function body is indented to indicate its scope.

Example:

```
def greeting(name):
    print(“Hello, “, name, “!”)
```

Using the **lambda** expression,

- This is a concise way to define a single-line function.
- It’s often used for simple functions that are only needed once.

Example:

```
square = lambda x: x*x

result = square(4)

print(result)  #Output: 16
```

Calling Function:

To call a function, you use its name followed by parentheses, passing any required arguments within the parentheses

Example:

```
#Calling a function

sum = add(5,9)

print(sum) #Output: 14
```

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

=> The **return** statement in Python is used to specify the value that a function should return to the caller. When a **return** statement is executed within a function, the function immediately terminates, and the specified value is passed back to the part of the code that called the function.

Some key points about the **return** statement;

- A function can have most one return statement.
- If a function doesn't have a return statement, it returns none.
- The return statement immediately terminates the function, so any code after it will not be executed.

Example:

```
def add(x, y):

        return x + y

result = add(3 + 5)

print(result)  #Output: 8
```

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

=> Iterators and Iterables are closely related concepts, but they have distinct characteristics:

Iterators;

Iterators in Python are objects that implement the **iter** and **next** methods. They provide a way to iterate over elements of a sequence, one at a time.

Key Differences:

- **Iteration mechanism:** Iterables provide the mechanism to iterate over their elements, while iterators are objects specifically designed for iteration.
- **Statefulness:** Iterators maintain their state (the current position in the sequence) during iteration, while iterables are generally stateless.
- **Multiple iterations:** Iterables can be iterated over multiple times, while iterators are typically used for a single iteration.

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

=> Generators in Python are special type of iterators that are defined using the **yield** keyword instead of return. They provide a way to create iterators that can pause and resume their execution, allowing for efficient memory usage and lazy evaluation.

Defining Generators:

To define generator, you use the def keyword followed by function name, parentheses for parameters (if any), and a colon. The generator function body contains yield statements to return values.

Example:

```
def count_up(r):
    for i in range(1, r+1):
        yield i


for number in count_up(5):
    print(number)
# Output
1
2
3
4
5
```

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

=> Here are some advantages of using generators over regular functions:

- **Lazy Evaluation:** Generators produce elements one at a time as needed, rather than generating all. This can be beneficial for large and infinite sequences to avoid unnecessary memory consumption.
- **Memory Efficient:** Generators can be more memory-efficient than lists or tuples, especially for large datasets. Since, they produce elements on-demand, they don't need to store the entire sequence in memory.
- **Infinite Sequences:** Generators can be used to create infinite sequences, which are not possible with regular functions.

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

=> Lambda functions, also known as anonymous functions, are a concise way to define small, unnamed functions in Python. They are often used for simple one-time operations or as arguments to higher-order functions.

Usages:

- Lambda functions are often used as arguments to functions that take other functions as input, such as map, filter, and reduce.
- If you need a small function that you'll only use once, a lambda function can be a concise and convenient way to define it.
- Lambda functions can be used within list comprehensions to create new lists based on existing ones.

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

=> The map() function in Python is a built-in function that applies a given function to each element of an iterable(list, tuple, string) and returns an iterator containing the results. It provides a concise and efficient way to transform elements of a sequence.

Usage:

- Applying a transformation to each element of a sequence.
- Combining multiple sequences element-wise.
- Creating new sequences based on existing ones.

Examples:

```
l = [1,2,3,4,5]
def sq(x):
    return x**2
list(map(sq, l)) #Using map
[1, 4, 9, 16, 25] #Output

# Function that add 10 to any number in the list
    list(map(lambda a: a + 10, l)) #Output: [11, 12, 13, 14, 15]
```

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

=>

Function	Purpose	Syntax	Example

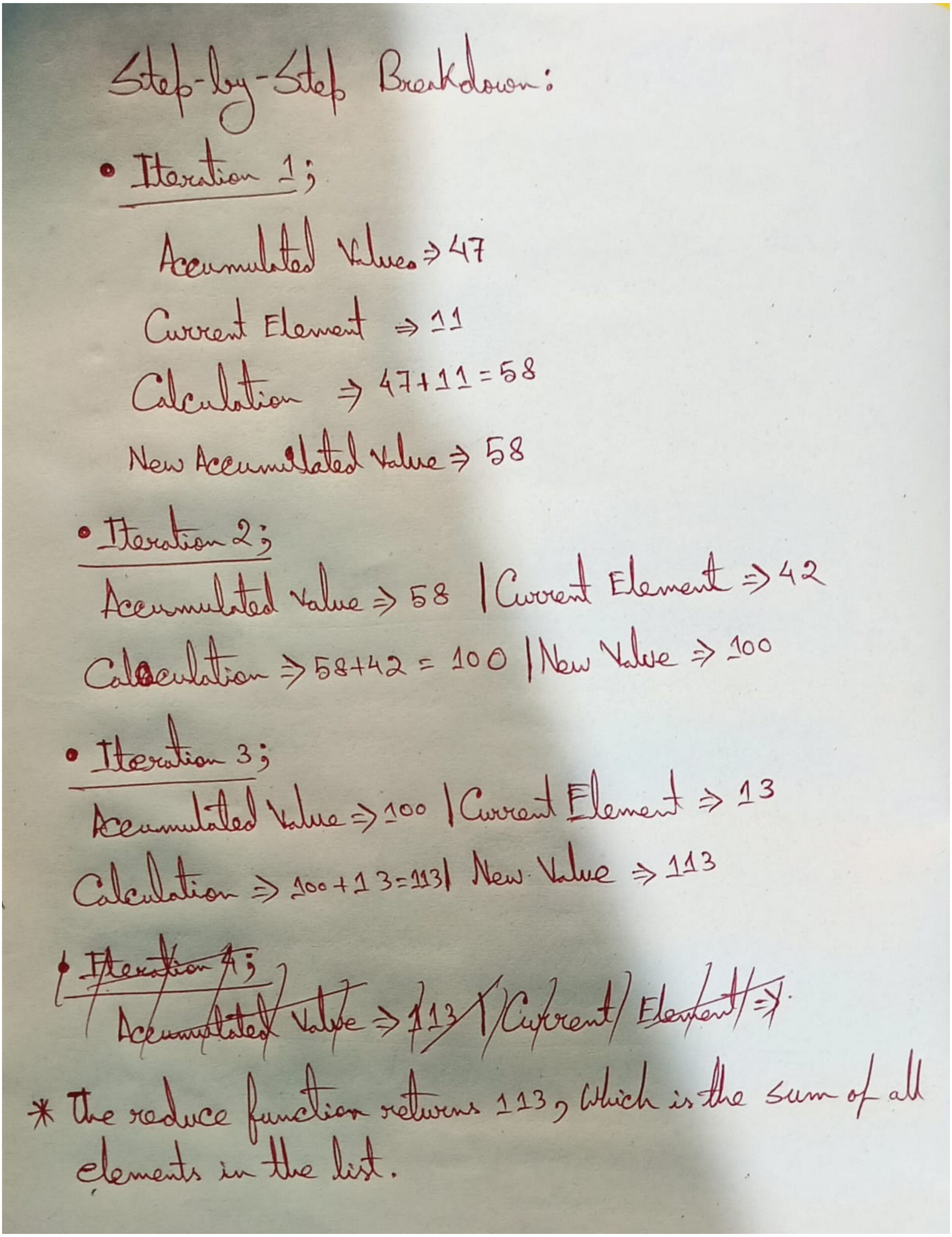
map()	Applies a function to every element of an iterable.	map(func, iterable)	# Function that add 10 to any number in the list:
			list(map(lambda a: a + 10, l))
			#Output:[11,12,13,14,15]

reduce()	Accumulates a single value from an iterable using a binary function.	reduce(function, iterable[, initializer])	#Find max number:
			max_numbers = [1,2,3,100,200,500,350,556,9,10,20]
			reduce(lambda x, y: x if x>y else y, max_numbers)
			#Output: 556

filter()	Filter elements from an iterable based on a predicate function.	filter(func/condition, iterable)	#Find even number:
			j = [1,2,3,4,5,6,7,8,9,10]
			list(filter(lambda x: x%2 == 0 , j))
			#Output: [2,4,6,8,10]

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

=>



Step-by-Step Breakdown:

- Iteration 1;

  Accumulated Values => 47

  Current Element => 11

  Calculation => 47+11=58

  New Accumulated Value => 58

- Iteration 2;

  Accumulated Value => 58 | Current Element => 42

  Calculation => 58+42 = 100 | New Value => 100

- Iteration 3;

  Accumulated Value => 100 | Current Element => 13

  Calculation => 100+13=113| New Value => 113

- ~~Iteration 4;~~

  ~~Accumulated Value => 113 | Current Element =>~~

* The reduce function returns 113, which is the sum of all elements in the list.

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.

=>

```
 """
Calculates the sum of even numbers in a list.
 Args:
   numbers: A list of numbers.
  Returns:
   The sum of even numbers in the list.
  """
        def sum_even_(numbers):
                sum_of_even = 0
                   for num in numbers:
                      if num % 2 == 0:
                          sum_of_even += num
                return sum_of_even

Usage:
        my_list = [1, 2, 3, 4, 5, 6]
        result = sum_even(my_list)
        print(result)  # Output: 12
```

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

=>

```
  """
```

```
Reverses a given string.


Args:
    string: The string to be reversed.


Returns:
    The reversed string.
  """
Usage:
            my_string = "Hello! Python"
            reversed_string = reverse(my_string)
            print(reversed_string)  # Output: dlrow olleh
```

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

=>

```
            """
            Squares each number in a list.
            Args:
             numbers: A list of numbers.
            Returns:
             A new list containing the squares of the numbers.
            """


def square_num(numbers):


  squared_nums = []
  for num in numbers:
    squared_nums.append(num ** 2)
  return squared_nums
```

Usage:

```
    num_list = [20, 30, 50, 10, 9]

    square_list = square_num(num_list)

    print(square_list) #Output: [400, 900, 2500, 100, 81]
```

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

=> # Checks if a given number is prime.

```
def prime(num):

  if num <= 1:
    return False
  if num <= 3:
    return True
  if num % 2 == 0 or num % 3 == 0:
    return False

  i = 5
  while i * i <= num:
    if num % i == 0 or num % (i + 2) == 0:
      return False
    i += 6

  return True
```

Usage:

```
    for num in range(1, 200):

    if prime(num):

    print(num, "is prime") #Output: It’ll show which numbers are prime.
```

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

```
=>      # Fibonacci series

        # Recursive function calls itself again and again


def gen_fib(n):

    if n <= 1:

      return n

    else:

      return gen_fib(n-1) + gen_fib(n-2)
```

Usage:

```
[gen_fib(n) for n in range (10)]

#Output: [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.

```
=>  """

        Generates powers of 2 up to a given exponent.

        Args:

        exponent: The maximum exponent.

        Yields:

        The powers of 2 up to the given exponent.

    """

def powers_of_two(exponent):


  power = 1

  for i in range(exponent + 1):

    yield power
    power *= 2
```

Usage:

```
exponent = 5
for power in powers_of_two(exponent):
  print(power)  #Output: 1, 2, 4, 8, 16, 32
```

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

=>

```
"""
Reads a file line by line and yields each line as a string.
    Args:
        filename: The name of the file to read.
    Yields:
        Each line of the file as a string.
 """
def read_file_line_by_line(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip() # .strip() removes any trailing newlines
```

Usage:

```
file_path = 'example.txt'

# Use the generator to read and print each line from the file
for line in read_file_line_by_line(file_path):
    print(line)
```

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

=>

```
#List of tuples

my_list = [(1, 'lion'), (3, 'lemon'), (2, 'tiger'), (4, 'dates')]


# Sort the list using a lambda function as the key

sorted_list = sorted(my_list, key=lambda x: x[1])


print(sorted_list)

 #Output: [(4, 'dates'), (3, 'lemon'), (1, 'lion'), (2, 'tiger')]
```

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

=>

```
# Converts Celsius to Fahrenheit.

def c_to_f(c):


  return (9/5) * c + 32


# Example list of temperatures in Celsius

celsius_temps = [-10, 10, 25, 40]


# Use map() to convert the list to Fahrenheit

fahrenheit_temps = list(map(c_to_f, celsius_temps))


print(fahrenheit_temps)

# Output: [14.0, 50.0, 77.0, 104.0]
```

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

=>

```
# Checking if a character is not a vowel
not_vowel = lambda char: char.lower() not in 'aeiouAEIOU'

# String
string = "Welcome, in the project section."

# Using filter() to remove vowels
filtered_string = ''.join(filter(not_vowel, string))

# Result
print(filtered_string)

#Output:  Wlcm, n th prjct sctn.
```

Q11. 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.

=>

```
def process_orders(orders):
  processed_orders = []
  for order in orders:
    order_number = order[0]
    quantity = order[2]
    price_per_item = order[3]
    total_price = quantity * price_per_item
    if total_price < 100:
      total_price += 10
    processed_orders.append((order_number, total_price))
  return processed_orders
# Usage:
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]
]

result = process_orders(orders)
print(result)
#Output:
[(34587, 163.8), (98762, 284.0), (77226, 108.85000000000001), (88112, 84.97)]
```