**Functions Assignment**

**Theory Questions:**

1.**Functions vs Methods**:

* Functions in python are independent self-defined block of code to perform a specific task and are called from anywhere in the code.
* Methods are defined to classes and can be called using dot operator(.)
* Functions offer code reusability and methods define the behaviour of classes.
* Functions operate on the data that is passed as arguments and methods operate on data of the object that is associated with.

Ex:

Function example,

def sum( a , b):

return a+b

a = 3

b = 4

result = sum(a,b)

print(result)

method example,

class example:

name = “suma”

def print\_name(self):

print(“MY name is : “,name)

example.print\_name = classmethod(example.print\_name)

example.print\_name

2.Function Arguments and Parameters in python:

**Function Arguments**: Function are those that are specified during the function call. These are the actual values that are being passed to the function based on which the function will evaluate and produce the output.

**Function Parameters:** Function parameters are those that specified during the function declaration. These are not the actual values.

Ex:

def sum( x , y):

Return x+y

a = 34

b = 56

print(sum(a,b))

In the above example,

x , y that are declared during the function declaration are called *parameters* and a , b that are passed during the function call are *arguments*.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. In Python, functions can be defined and called in several ways depending on the use case. Here are different methods:

**1. Regular Function Definition**

A basic function is defined using the def keyword, followed by the function name and parameters .

def greet(name):

return f"Hello, {name}!"

# Calling the function

print(greet("Alice"))

**2. Function with Default Parameters**

You can set default values for parameters, making them optional.

def greet(name="Guest"):

return f"Hello, {name}!"

# Calling the function with and without arguments

print(greet("Bob"))

print(greet())

**3. Lambda (Anonymous) Functions**

Lambda functions are small, unnamed functions that are used for short-term purposes. They're defined using the lambda keyword.

greet = lambda name: f"Hello, {name}!"

# Calling the lambda function

print(greet("Charlie"))

**4. Function with Arbitrary Arguments (\*args)**

If you don’t know how many arguments will be passed, we use \*args to collect them in a tuple.

def greet\_all(\*names):

return [f"Hello, {name}!" for name in names]

# Calling the function with multiple arguments

print(greet\_all("Alice", "Bob", "Charlie"))

**5. Function with Keyword Arguments (\*\*kwargs)**

For named arguments, we use \*\*kwargs, which collects them into a dictionary.

def greet\_with\_title(\*\*kwargs):

return [f"Hello, {title} {name}!" for name, title in kwargs.items()]

# Calling the function with keyword arguments

print(greet\_with\_title(Alice="Ms.", Bob="Dr."))

**6. Function as First-Class Citizens**

In Python, functions can be passed as arguments to other functions.

def welcome\_message(func, name):

return func(name)

# Defining a basic function

def greet(name):

return f"Hello, {name}!"

# Passing 'greet' as an argument

print(welcome\_message(greet, "Dana"))

**7. Nested Functions**

A function can be defined inside another function.

def outer\_function(name):

def inner\_function():

return f"Hello, {name}!"

return inner\_function()

# Calling the nested function

print(outer\_function("Eve"))

The above are some of the common ways to define and call functions in Python.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4 . **Return Statement in python:**

The return statement is used to exit a function and send back a value (or values) to the caller.

We use return statement for:

* Outputting a result a function.

Ex: def add(a, b):

return a + b

result = add(5, 3)

print(result) # Outputs: 8

* Exiting a function early

Ex: def check\_number(num):

if num < 0:

return "Negative"

return "Positive"

print(check\_number(-5)) # Outputs: Negative

print(check\_number(5)) # Outputs: Positive

* Returning multiple values

Ex: def get\_coordinates():

x = 10

y = 20

return x, y

coords = get\_coordinates()

print(coords) # Outputs: (10, 20)

* Returning nothing i.e, returning None
* Enabling function reusability

Ex: def multiply(a, b):

return a \* b

result1 = multiply(4, 5) # Can be used multiple times

result2 = multiply(2, 3)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 5. ****Iterators****:

An **iterator** is an object that enables traversing through all the elements of a collection (such as lists, tuples, dictionaries, or sets) one element at a time. It consists of two main methods:

1. \_\_iter\_\_(): Returns the iterator object itself.
2. \_\_next\_\_(): Returns the next element from the collection and raises StopIteration when there are no more elements.

my\_list = [1, 2, 3]

iterator = iter(my\_list)

print(next(iterator)) # Outputs: 1

print(next(iterator)) # Outputs: 2

print(next(iterator)) # Outputs: 3

# Calling next() again will raise StopIteration

* An iterator keeps track of where it is in the sequence.
* Once an element has been traversed, it cannot be revisited unless you reinitialize the iterator.

### ****Iterables**** :

An **iterable** is any Python object that can return an iterator when the iter() function is called on it. This includes collections like lists, tuples, strings, and sets, among others.

my\_list = [1, 2, 3] # A list is an iterable

for item in my\_list:

print(item) # Outputs: 1, 2, 3

* An iterable is an object that contains or can generate elements one by one.
* It is not an iterator by itself, but it can return an iterator using the iter() function.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. **Generators** are a special type of iterable, like lists or tuples, but unlike them, generators don’t store their contents in memory. Instead, they generate items on the fly, making them more memory efficient, especially for handling large datasets.

**Key Concepts of Generators:**

1. **Lazy Evaluation**: Generators produce items one at a time, only when requested. This is useful for large data sequences or streams, as it saves memory and improves performance.
2. **Yield Statement**: Generators are defined using the yield keyword instead of return. Each time yield is called, the function’s state is saved, and execution is paused until the next item is requested.

**Defining Generators**

generators in Python can be defined using a function that contains one or more yield statements.

def simple\_generator():

yield 1

yield 2

yield 3

gen = simple\_generator()

for value in gen:

print(value)

When the generator is called, it doesn’t execute the function right away. Instead, it returns a generator object. On each iteration, the function executes until it encounters yield, where it pauses and returns the value. The next time the generator is called, it resumes where it left off.

**Generator Expressions**

Generators can also be created using expressions, similar to list comprehensions but with parentheses instead of square brackets:

gen\_exp = (x \* x for x in range(5))

for num in gen\_exp:

print(num)

Advantages:

1. **Memory Efficiency**: Generators only compute values when needed, which is beneficial for large datasets.
2. **Single Iteration**: Once a generator is exhausted (i.e., all values have been generated), it cannot be reused unless it's explicitly recreated.

Generators are highly useful in scenarios where memory efficiency or processing streams of data is important.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7. Generators in Python have several advantages over regular functions:

1. **Memory Efficiency**: Generators produce values one at a time using yield, without storing the entire dataset in memory. Regular functions return all data at once, which can be memory-intensive, especially for large datasets.
2. **Lazy Evaluation**: Generators compute values only when needed, whereas regular functions calculate everything upfront. This makes generators more efficient, particularly when only part of the dataset is required.
3. **Handling Infinite Sequences**: Generators can work with infinite data streams (e.g., real-time sensor data) since they yield values as needed. Regular functions can't handle infinite sequences.
4. **Improved Performance**: Generators spread processing over time, allowing faster initial responses since values are produced on-the-fly, unlike regular functions that process everything before returning.
5. **Simplified State Management**: Generators automatically maintain their state between iterations, making them ideal for complex iterative processes without additional code.
6. **Pipeline Processing**: Generators enable efficient, memory-saving pipelines by processing data in stages, passing one value at a time, while regular functions handle the entire dataset at once.
7. **I/O Efficiency**: Generators can handle large I/O operations (e.g., reading a file) incrementally, improving performance over regular functions, which read all data into memory upfront.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

8. A **lambda function** in Python is a small, anonymous function defined using the lambda keyword. It can take multiple arguments but is limited to a single expression. The result of the expression is automatically returned.

**Syntax:**

lambda arguments: expression

Ex:

add = lambda x, y: x + y

print(add(2, 3)) # Output: 5

**When to Use Lambda Functions:**

1. **With functions like map(), filter(), and reduce()**:
   * For applying quick, simple operations to collections.

Example with filter():

even\_numbers = list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4]))

1. **For short, simple callbacks**:
   * Often used in sorting or event-driven programming.

Example with sorted():

sorted\_list = sorted([(1, 2), (3, 1)], key=lambda x: x[1])

1. **Inline use**:
   * When a small, temporary function is needed for a quick task.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

9. The map() function in Python is used to apply a given function to **each item of an iterable** (like a list, tuple, etc.) and return a map object (which is an iterator) containing the results. The map() function allows you to **process elements** of a sequence without needing to explicitly write a loop, making the code more concise and readable.

**Syntax:**

map(function, iterable, ...)

* **function**: A function that defines the operation to apply to each element.
* **iterable**: One or more iterables (like lists or tuples) on which the function is applied.

**Purpose of using map() funtion:**

1. **Apply a Function to Each Element**: The primary purpose of map() is to apply a function to every item in an iterable and return the results.

**Example**: Squaring each number in a list.

numbers = [1, 2, 3, 4]

result = map(lambda x: x \* x, numbers)

print(list(result)) # Output: [1, 4, 9, 16]

1. **Simplifies Code**: It allows you to transform iterables with fewer lines of code compared to a for loop.

**Using map()**:

result = map(str.upper, ['apple', 'banana', 'cherry'])

print(list(result)) # Output: ['APPLE', 'BANANA', 'CHERRY']

**Using a for loop** (longer version):

result = []

for fruit in ['apple', 'banana', 'cherry']:

result.append(fruit.upper())

print(result) # Output: ['APPLE', 'BANANA', 'CHERRY']

1. **Handles Multiple Iterables**: map() can process multiple iterables simultaneously, applying the function to corresponding elements.

**Example**: Adding two lists element-wise.

list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = map(lambda x, y: x + y, list1, list2)

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

### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 10.

### map(), filter(), and reduce() are built-in Python functions used for functional programming. They allow you to process iterables (like lists) efficiently.

**1. map():**Applies a function to **each item** of an iterable and returns a new iterable (usually converted to a list).

**Use Case**: Transforming or modifying all elements in an iterable.

# Example: Squaring all numbers in a list

numbers = [1, 2, 3, 4]

squared = list(map(lambda x: x \* x, numbers))

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

**2. filter():** Filters elements of an iterable based on a **condition** (i.e., returns only the items for which the condition is True).

**Use Case**: Selecting certain elements from a collection based on a condition.

# Example: Filtering even numbers from a list

numbers = [1, 2, 3, 4]

evens = list(filter(lambda x: x % 2 == 0, numbers))

print(evens) # Output: [2, 4]

**3. reduce():**

Applies a function **cumulatively** to the elements of an iterable, reducing it to a single value. It must be imported from functools.

**Use Case**: Combining all elements into a single result (e.g., summing all numbers).

from functools import reduce

# Example: Summing all numbers in a list

numbers = [1, 2, 3, 4]

total = reduce(lambda x, y: x + y, numbers)

print(total) # Output: 10

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

11.

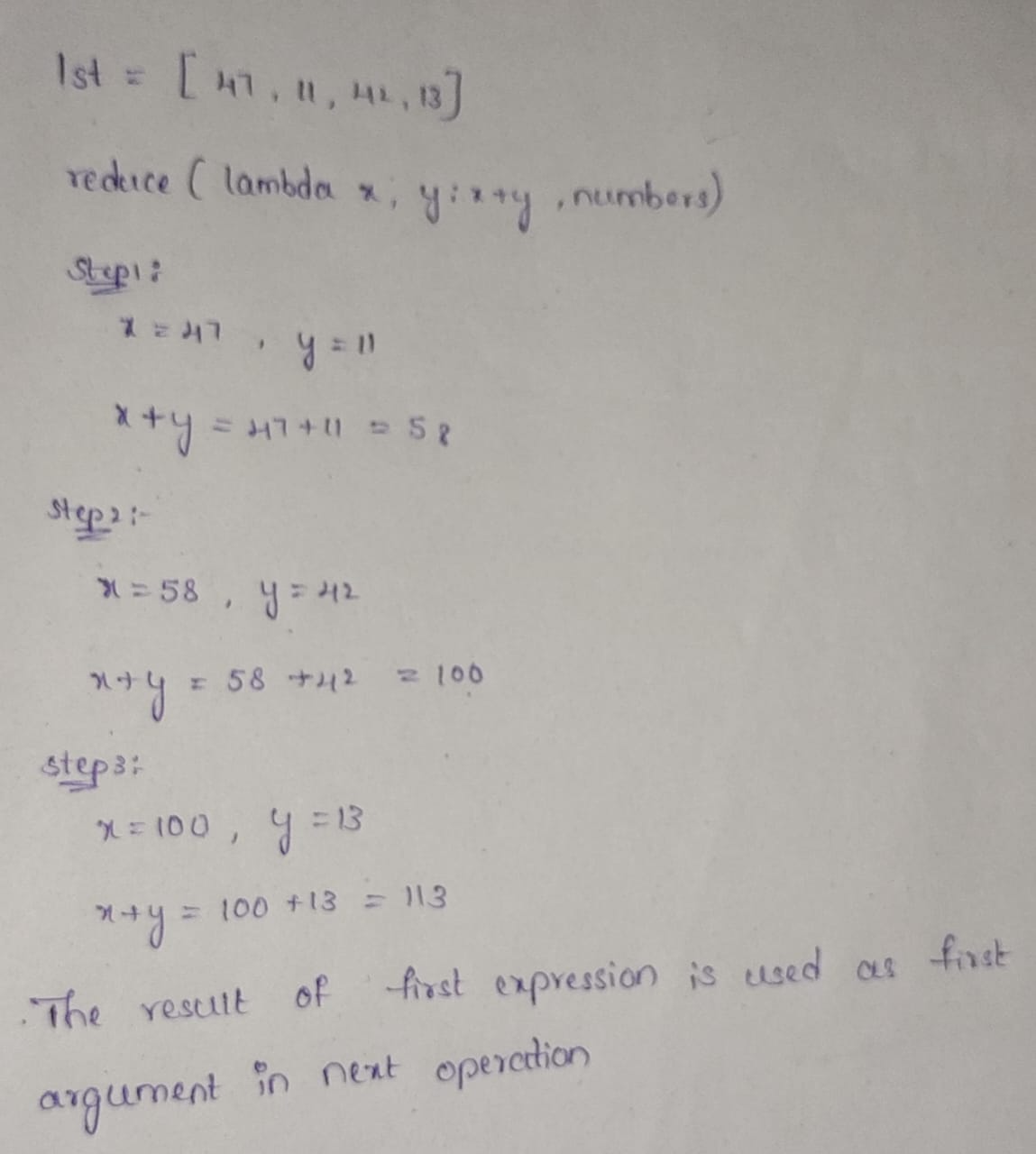
# source code :

from functools import reduce

numbers = [47, 11, 42, 13]

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

print(result)



Bottom of Form