Functions Assignment

Data Science with Gen AI

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

Ans:- In Python, **functions** and **methods** are both blocks of reusable code, but they differ in their definition, usage, and context.

Method is a term that typically used in Object-oriented context (like in Java)

- Function is defined independentaly, while method is defined inside a class.
- Functions can be called independently as it is not associated with any object, while method will always be called with its associated object.

Example:

Function

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

Method

```
class Greeter:
    def greet(self, name):
        return f"Hello, {name}!"
obj = Greeter()
obj.greet("Anoop")
```

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

Ans:-

Function Argument

- Arguments are the values that are passed to the function when it is called.
- They replace the parameters during execution.

Function Parameters

- Parameters are the variables writtent inside the parentheses in the function definition.
- They act as placeholders for the values that the function will receive when called.

Example

```
def greet(name, message): # name and message are parameters
    print(f"{message}, {name}!")
greet("Anoop", "Hello") # "Anoop" and "Hello" are arguments
```

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

Ans:-

There are different ways of defining a function.

- No parameters

```
def say_hello():
    print("Hello, World!")
```

- With parameters

```
def add(a, b):
    return a + b
```

- Default parameters

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

- Variable-Length Arguments (*args)

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

- Keyword Arguments (**kwargs)

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

- Lambda (Anonymous) Functions

```
square = lambda x: x ** 2
```

There are different ways of calling a function.

- Basic call

```
result = function_name(arguments)
```

- Positional Arguments (Order Matters)

```
def add(a, b):
    return a + b
result = add(10, 5) # 10 → a, 5 → b → Output: 15
```

- Keyword Arguments (Order Doesn't Matter)

```
result = add(b=5, a=10) # Same as add(10, 5)
```

- Mixing Positional & Keyword Arguments (Positional arguments must come before keyword arguments.)

```
result = subtract(10, b=5) # Valid
    #result = subtract(a=10, 5) # Invalid (SyntaxError)
```

- Using Default Parameters

```
greet() # Uses default → "Hello, Guest!"
greet("Anoop") # Overrides default → "Hello, Anoop!"
```

- Passing *args (Unpacking Iterables)

```
numbers = [3, 5]
result = add(*numbers) # Equivalent to add(3, 5) → 8
```

- Passing **kwargs (Unpacking Dictionaries)

```
details = {"a": 10, "b": 5}
result = subtract(**details) # Equivalent to subtract(a=10, b=5) → 5
```

- Calling Lambda Functions

```
square = lambda x: x ** 2
print(square(4)) # Output: 16
```

Q. (4) What is the purpose of the "return" statement in a Python function?

Ans:- The return statement in a Python function has two key purposes.

- It terminates the function
- Sends some value back to the calling function

```
def add(a, b):
    return a + b # Exits here and returns the sum
    print("This won't execute") # Skipped
result = add(3, 5) # result = 8
```

It can return multiple values as Tuple. If no return statement is written in the function, it returns *None*.

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

Ans:-

Iterables -An object that can be looped over and over (e.g., lists, tuples, strings, dictionaries, sets).

- It implements the **iter**() method (or **getitem**() for backward compatibility), which returns an iterator.

```
my_list = [1, 2, 3] # List (iterable)
```

Iterators

- An object that represents a stream of data.
- Implements **iter**() (returns itself) and **next**() (returns the next item or raises StopIteration when exhausted).

```
my_iter = iter([1, 2, 3]) # Converts list to iterator
print(next(my_iter)) # 1
print(next(my_iter)) # 2
```

print(next(my_iter)) # 3
print(next(my_iter)) # Raises StopIteration

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

Ans:-
Generators are a special type of iterator that allow us to generate values easily instead of storing them all in memory at once. They are memory-efficient and ideal for handling large collection of data.

- Generators yield values one at a time using the yield keyword.
- Unlike regular functions that use *return* (which exits the function), generators pause execution at *yield* and resume where they left off when next called.
- They automatically implement the iterator methods like (**iter**() and **next**()), so the program can loop over them.

def count_up_to(max_num):

> num = 1
> while num <= max_num:
>
> > yield num # Pauses here and returns num
> > num += 1

counter = count_up_to(3)
print(next(counter)) # 1
print(next(counter)) # 2
print(next(counter)) # 3

> #print(next(counter)) # Raises StopIteration

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

Ans:-
Generators provide some benefits compared to regular functions in some cases, particularly when dealing with large data collection, streams of data, or memory-intensive operations.

- It is memory efficient
- On-Demand Computation (In regular functions, all computation is done at one go, while in Generators, value is computed as and when required.)
- Generators avoid unnecessary computations.
- Generators can be chained to create efficient data pipelines.
- Generators remember their state between *yield* statements.

Q. (8) What is a lambda functions in Python and when is it typically used?

Ans:-
A lambda function (or anonymous function) is a small, single-expression function defined using the *lambda* keyword. Unlike regular functions (defined with *def*), lambdas are unnamed and are typically used for short, simple operations.

square = lambda x: x ** 2
print(square(5)) # Output: 25

Q. (9) Explain the purpose and use of "map()" function in Python.

Ans:-
The map() function is a built-in Python tool for applying a given function to every item in an iterable (like a list, tuple, or string) and returning an iterator that yields the transformed results. It's a key tool for functional programming in Python, enabling the programmer to write compact code for efficient data processing without loops.

str_numbers = ["1", "2", "3"]
int_numbers = map(int, str_numbers) # Applies `int()` to each item
print(list(int_numbers)) # Output: [1, 2, 3]

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

Ans:-
These three functions are core tools in Python's functional programming paradigm, but they serve distinct purposes:

map() -> Transforms each item in an iterable
filter() -> Selects items that meet a condition
reduce() -> Aggregates items into a single value

map()
Applies a function to every item in an iterable and returns a new iterator with the transformed values.

numbers = [1, 2, 3]
squared = map(lambda x: x ** 2, numbers)
print(list(squared)) # Output: [1, 4, 9]

filter()

Selects items from an iterable only if they satisfy a condition (specified by the function).

numbers = [1, 2, 3, 4, 5]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens)) # Output: [2, 4]

reduce()

Combines all items in an iterable into a single value by repeatedly applying a function.

from functools import reduce
numbers = [10, 20, 30, 40]
sum_result = reduce(lambda x, y: x + y, numbers)
print(sum_result) # Output: 100 (10+20+30+40)

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

```
# Q(1) 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_even(num):
  sum = 0
  for i in num:
    if i%2 == 0:
      sum = sum+i
  return sum

lst1 = [1,2,3,4,5,6,7,8,9]
result = sum_even(lst1)
print(result)
```

```
20
```

```
# Q(2) Create a Python function that accepts a string and returns the reverse of
# that string.

def reverse_str(str):
  return str[::-1]

s = "Anoop Verma"
result = reverse_str(s)
print(result)
```

```
amreV poonA
```

```
# Q(3) Implement a Python function that takes a list of integers and returns a
# new list containing the squares of each number.

def sq_num(nums):
  squared_num = []
  for n in nums:
    squared_num.append(n ** 2)
  return squared_num

lst1 = [1,2,3,4,5,6,7,8,9]
result = sq_num(lst1)
print(result)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
# Q(3) - Using map() and lambda()

def sq_num(nums):
  return list(map(lambda x: x ** 2, nums))

lst1 = [10,20,30,40,50,60,70,80,90]
result = sq_num(lst1)
print(result)
```

```
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]
```

```
# Q(4) Write a Python function that checks if a given number is prime or not
# from 1 to 200.

def is_prime(n):
  if n <= 1:
    return False
  if n == 2:  # 2 is the only even prime
    return True
  if n % 2 == 0:  # Eliminate even numbers
    return False
  # Check divisibility up to sqrt(n) (odd numbers only)
  for i in range(3, int(n**0.5) + 1, 2):
```

```
  for i in range(3, int(n**0.5) + 1, 2):
    if n % i == 0:
      return False
    return True

for num in range(1, 201):
  if is_prime(num):
    print(num, end=',')
```

```
2,11,13,17,19,23,25,29,31,35,37,41,43,47,49,53,55,59,61,65,67,71,73,77,79,83,85,89,91,95,97,101,103,107,109,113,115,119,121,125,127,131,133,137,13
```

```
# Q(5) Create an iterator class in Python that generates the Fibonacci sequence up to a
# specified number of terms.

class FibonacciIterator:
    def __init__(self, max_terms):
        self.max_terms = max_terms
        self.current_term = 0
        self.a, self.b = 0, 1  # Initialize Fibonacci sequence

    def __iter__(self):
        return self

    def __next__(self):
        if self.current_term >= self.max_terms:
            raise StopIteration
        value = self.a
        self.a, self.b = self.b, self.a + self.b  # Update Fibonacci values
        self.current_term += 1
        return value

fib = FibonacciIterator(25)  # Generate first 25 Fibonacci numbers
for num in fib:
    print(num, end=" ")
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368
```

```
# Q(6) Write a generator function in Python that yields the powers of 2 up to a
# given exponent.

def powers_of_2(max):
    power = 1  # 2^0 = 1
    for _ in range(max + 1):
        yield power
        power *= 2

for exponent, result in enumerate(powers_of_2(10)):
    print(f"2^{exponent} = {result}")
```

```
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
```

```
# Q(7) Implement a generator function that reads a file line by line and yields
# each line as a string.

def read_line(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()  # Remove trailing newline characters

for line in read_line('Example.txt'):
    print(line)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-10-3342040937> in <cell line: 0>()
      7             yield line.strip()  # Remove trailing newline characters
      8
----> 9 for line in read_line('Example.txt'):
```

Next steps: Explain error

```
# Q(8) Use a lambda function in Python to sort a list of tuples based on the
# second element of each tuple.

# Original list of tuples
data = [('Anoop', 3), ('Ajit', 1), ('Ashok', 2), ('Amitabh', 5), ('Abhishek', 4)]

# Sort using lambda function
sorted_data = sorted(data, key=lambda x: x[1])

print("Original data")
print(data)
print("Sorted data")
print(sorted_data)
```

```
Original data
[('Anoop', 3), ('Ajit', 1), ('Ashok', 2), ('Amitabh', 5), ('Abhishek', 4)]
Sorted data
[('Ajit', 1), ('Ashok', 2), ('Anoop', 3), ('Abhishek', 4), ('Amitabh', 5)]
```

```
# Q(9) Write a Python program that uses `map()` to convert a list of
# temperatures from Celsius to Fahrenheit.

# List of temperatures in Celsius
celsius = [0, 10, 20, 30, 40, 50, 60, 70, 90, 100]

# Conversion formula: F = (C × 9/5) + 32
fahrenheit = list(map(lambda c: (c * 9/5) + 32, celsius))

# Display results
print("Celsius:", celsius)
print("Fahrenheit:", fahrenheit)
```

```
Celsius: [0, 10, 20, 30, 40, 50, 60, 70, 90, 100]
Fahrenheit: [32.0, 50.0, 68.0, 86.0, 104.0, 122.0, 140.0, 158.0, 194.0, 212.0]
```

```
# Q(10) Create a Python program that uses `filter()` to remove all the vowels
# from a given string.

def rem_vowels(in_str):
    vowels = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'}
    filtered = filter(lambda char: char not in vowels, in_str)
    return ''.join(filtered)

text = "Hello, World! My name is Anoop Verma"
result = rem_vowels(text)
print(result)
```

```
Hll, Wrld! My nm s np Vrm
```

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

# 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.
```