**Assignment 4 :-**

Q1.

**Key Differences**

| **Aspect** | **Required Argument** | **Default Argument** |
| --- | --- | --- |
| Must be provided | ✅ Yes | ❌ No (uses default if not passed) |
| Position in function | Comes **before** default arguments | Must be placed **after** required ones |
| Example | def func(x) | def func(x=10) |

def func(x=5):

print(x)

func() # prints 5 (default)

func(None) # prints None (explicit value overrides default)

def greet(name, greeting="Hello"):

return f"{greeting}, {name}!"

# Example usage:

print(greet("Alice")) # "Hello, Alice!"

print(greet("Bob", "Good morning")) # "Good morning, Bob!"

print(greet("Eve", None)) # "None, Eve!" (explicit None overrides default)

def greet(name, greeting=None):

if greeting is None:

greeting = "Hello"

return f"{greeting}, {name}!"

print(greet("Eve")) # "Hello, Eve!"

print(greet("Eve", None)) # Still "Hello, Eve!"

Q2.

**\*args – Variable Positional Arguments**

* Collects **extra positional arguments** into a **tuple**.
* Used when you **don’t know how many arguments** will be passed.

def example(\*args):

print(args)

example(1, 2, 3) # Output: (1, 2, 3)

**🔸 \*\*kwargs – Variable Keyword Arguments**

* Collects **extra keyword arguments** into a **dictionary**.
* Useful for passing **optional configuration-style** parameters.

def example(\*\*kwargs):

print(kwargs)

example(a=1, b=2) # Output: {'a': 1, 'b': 2}

**Using \*args and \*\*kwargs Together**

You can use both in the same function, **\*args must come before \*\*kwargs**:

def func(\*args, \*\*kwargs):

print(args)

print(kwargs)

def summarize(\*args, \*\*kwargs):

total = sum(args)

if kwargs.get("square"):

total = total \*\* 2

if kwargs.get("negate"):

total = -total

return total

print(summarize(1, 2, 3)) # ➝ 6

print(summarize(1, 2, 3, square=True)) # ➝ 36

print(summarize(1, 2, 3, negate=True)) # ➝ -6

print(summarize(1, 2, 3, square=True, negate=True)) # ➝ -36

Q3.

**Pass-by-Value vs Pass-by-Reference**

These are two common ways programming languages handle function arguments:

| **Concept** | **Description** |
| --- | --- |
| **Pass-by-Value** | A **copy** of the variable is passed — changes **do not** affect the original. |
| **Pass-by-Reference** | A **reference (memory address)** to the original object is passed — changes **do** affect the original. |

def modify\_list(my\_list):

my\_list.append("new item")

print("Inside function:", my\_list)

# Original list

original = [1, 2, 3]

modify\_list(original)

print("Outside function:", original)

output:-

Inside function: [1, 2, 3, 'new item']

Outside function: [1, 2, 3, 'new item']

Q4.

A **decorator** is a function that **takes another function as input**, adds some functionality **before or after** its execution, and **returns a new function** — *without modifying the original function’s code*.

It's a powerful and elegant way to add **logging, timing, authentication, etc.**, to functions.

def my\_decorator(func):

def wrapper():

print("Something before the function runs")

func()

print("Something after the function runs")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

import time

def timing(func):

def wrapper(\*args, \*\*kwargs):

start = time.time()

result = func(\*args, \*\*kwargs)

end = time.time()

print(f"⏱️ Function '{func.\_\_name\_\_}' executed in {end - start:.4f} seconds")

return result

return wrapper

output:-

Finished slow task

⏱️ Function 'slow\_function' executed in 2.0001 seconds

Q5.

**Generators** are **special functions** that return an **iterator** and **yield values one at a time** using the yield keyword instead of return.

**Difference from Regular Functions**

| **Feature** | **Regular Function** | **Generator Function** |
| --- | --- | --- |
| Return Type | Returns a single result | Yields multiple values one by one |
| Memory Usage | Loads all values into memory | Produces values **lazily** (on demand) |
| Performance (for large data) | May be slow or memory-heavy | Faster and memory-efficient |
| State Retention | No memory between calls | **Remembers** its state between yields |
| def countdown(n):  while n > 0:  yield n  n -= 1  for number in countdown(5):  print(number)  5  4  3  2  1  Q6.  **yield vs return**   | **Feature** | **return** | **yield** | | --- | --- | --- | | Stops execution | Yes – exits the function entirely | No – pauses execution, keeps local state | | Produces values | Only once | Multiple times (lazy, one at a time) | | Memory usage | Returns all values at once | Generates values on demand (memory-efficient) | | Use case | Final result | Streams, sequences, large data sets |   def demo\_return():  return 1  return 2 # This is never reached  def demo\_yield():  yield 1  yield 2 # Will be reached in next iteration  def fibonacci():  a, b = 0, 1  while True:  yield a  a, b = b, a + b  fib = fibonacci()  for \_ in range(10):  print(next(fib))  output:-  0  1  1  2  3  5  8  13  21  34  Q7.  Generators are **memory-efficient** tools in Python. Instead of loading all data into memory (like a list does), generators:   * **Yield one item at a time**, * **Pause execution between items**, * Handle **huge files or data streams** without running out of RAM.   **Example Use Cases:**   * Reading large files (logs, datasets). * Streaming data from web APIs. * Processing sensor or user input data in real-time. * Efficient data pipelines.   Q8.  A **generator expression** is similar to a **list comprehension**, but instead of creating and storing the entire list in memory, it **yields items one by one** — just like a generator function using yield.  **Generator Expressions vs List Comprehensions**   | **Feature** | **List Comprehension** | **Generator Expression** | | --- | --- | --- | | Syntax | [x for x in iterable] | (x for x in iterable) | | Output | A full list in memory | A generator object (lazy evaluation) | | Memory Usage | High for large data sets | Low — generates one item at a time | | Performance | Faster access, but higher memory usage | Slower access, better for large data | | Use When | You need the entire list immediately | You process items one by one (e.g., in a loop) |   squares = [x \*\* 2 for x in range(1000000)]  squares = (x \*\* 2 for x in range(1000000))  for i in squares:  if i > 100:  break  print(i)  Q9.  Lambda functions are **anonymous**, **inline** functions defined with the lambda keyword.  **Syntax:**  python  CopyEdit  lambda arguments: expression  They're typically used for **short, simple operations**, especially in places where a full def function would feel too verbose.  **When to Use Lambda Functions**  Use lambda functions when:   * You need a short function **once** or **in-place**. * You’re working with higher-order functions like map(), filter(), or sorted().   **Limitations of Lambda Functions**   | **Limitation** | **Explanation** | | --- | --- | | Only one expression | No statements (e.g., no if, for, etc.) | | Harder to read/debug | Anonymous and compact | | Cannot contain docstrings | Limited for documentation | | No annotations | Can't easily define types or metadata |   multiply = lambda x, y: x \* y  list1 = [1, 2, 3, 4]  list2 = [10, 20, 30, 40]  result = list(map(multiply, list1, list2))  print(result) # Output: [10, 40, 90, 160]  Q10.  **map() + lambda**  map(func, iterable) applies func to each item in the iterable.  python  CopyEdit  words = ['hello', 'world', 'python']  upper\_words = list(map(lambda x: x.upper(), words))  print(upper\_words) # Output: ['HELLO', 'WORLD', 'PYTHON']  **2. filter() + lambda**  filter(func, iterable) keeps items **where func(item) returns True**.  python  CopyEdit  numbers = [1, 2, 3, 4, 5, 6]  odd\_numbers = list(filter(lambda x: x % 2 != 0, numbers))  print(odd\_numbers) # Output: [1, 3, 5]  **3. reduce() + lambda**  reduce(func, iterable) **reduces** the iterable to a single value by applying the function cumulatively.  python  CopyEdit  from functools import reduce  nums = [1, 2, 3, 4]  product = reduce(lambda x, y: x \* y, nums)  print(product) # Output: 24 |  |  |