Editorial-Approach-W3A1: Mutable Defaults, Higher-Order Functions, and File Operations

---

Question 1

What will be the output of the following code?

```
def func(a, b=[]):
    b.append(a)
    return b

print(func(1))
print(func(2))
print(func(3))
```

Options: A)

[1]

[2]

[3]

B)

[1]

[1, 2]

[1, 2, 3]

C)

[1]

[2]

[3, 2, 1]

D)

[1]

[1]

[1]

Approach:

1. **Observe the Default Argument**: Notice that b=[] is a default parameter. Recall that in Python, default parameters are **only evaluated once** when the function is defined.

2. **Recognize Mutability**: A list ([]) is a **mutable** object. Think about how mutability might affect the contents of b across multiple calls.
3. **Check Each Call**: Call the function step by step:
    - First call: How does the list look after appending the first value?
    - Second call: Does the list reinitialize or continue from the previous state?
    - Third call: What does the list contain now?
4. **Compare What You Expect**: If the same list is reused, how do the appended elements accumulate?

Question 2

What will be the output of the following code?

```
def outer(x):
    def inner(y):
        return x + y
    return inner

add_five = outer(5)
print(add_five(10))
```

Options: A) 5

B) 10

C) 15

D) Error

Approach:

1. **Identify the Nested Function**: Notice how outer returns inner.
2. **Recognize Closures**: inner still has access to x (from outer) even after outer has finished executing. This is a key point of closures in Python.
3. **Focus on x**: When you do outer(5), think about what value of x the returned function inner will remember.
4. **Call the Returned Function**: add_five(10) effectively uses that remembered value of x plus the new argument y.
5. **Perform the Arithmetic**: Evaluate the expression in inner using the captured value of x.

Question 3

What will be the output of the following code?

```
def func(x):
    return x * 2


lst = [1, 2, 3, 4]
result = list(map(lambda x: func(x), lst))
print(result)
```

Note:

- The map function applies a given function to each item in an iterable (in this case, the list lst).
- The lambda x: func(x) is an anonymous function that calls func(x) for each element x in lst.

Options: A) [2, 4, 6, 8]

B) [1, 2, 3, 4]

C) [1, 4, 9, 16]

D) Error

Approach:

1. **Understand map**: map(some_function, iterable) applies some_function to each item in iterable.
2. **Check the Function Being Mapped**: func(x) takes x and multiplies by 2.
3. **Combine map with lambda**: Even though the lambda just calls func(x), note that each element of lst will be processed.
4. **Trace Through Each Element**:
    - Take the first element from lst, apply func.
    - Take the second element, apply func.
    - ... and so on.
5. **Convert to List**: Remember map returns an iterator in Python 3, so list(...) collects all processed values into a list.

Question 4

What will be the output of the following code?

```
def func(a, b, c):
    return a + b + c


values = (1, 2, 3)
print(func(*values))
```

Options: A) 6

B) (1, 2, 3)

C) Error

D) None

Approach:

1. **Look at the Function Signature**: It requires three parameters.
2. **Inspect the Argument Unpacking**: *values unpacks the tuple (1, 2, 3) into three separate arguments.
3. **Rewrite Mentally**: func(*values) is like calling func(values[0], values[1], values[2]).
4. **Perform the Operation**: The function then combines the three numbers (in whatever way the code specifies).
5. **Watch for the Result**: The printed output will be the sum (or combination) of these three arguments.

For Question 5 - 8: Please refer to the Kartik's Sir class Google Colab: Link to the Colab

Question 5:

What is the purpose of the game() function in the provided code?

Options:

A) To simulate a battle between two players with random attacks and defenses.

B) To calculate the probability of winning for each player.

C) To generate random numbers for a dice game.

D) To create a graphical user interface for a game.

Question 6:

What happens when a player chooses to defend in the game() function?

Options:

A) The player's health is fully restored.

B) The player takes half damage during the opponent's attack.

C) The player's next attack deals double damage.

D) The player's health is reduced by half.

Question 7:

What is the role of the computer_choice() function in the code?

Options:

A) It randomly selects between attack and defend for Player 1.

B) It determines the optimal move for Player 2 based on the game state.

C) It calculates the total damage dealt by both players.

D) It ends the game when a player's health reaches 0.

Question 8:

What is the significance of the turn variable in the game() function?

Options:

A) It keeps track of the total number of turns played in the game.

B) It determines which player's turn it is to attack or defend.

C) It calculates the remaining health of both players.

D) It decides the winner of the game.

Question 9

What will be the output of the following code?

```
f1 = open("student.txt", "w")
f1.write("Hello, World!")
f1.close()


f1 = open("student.txt", "r")
print(f1.read(5))
f1.close()
```

Note: student.txt file never existed before.

Options: A) Hello

B) Hello,

C) World

D) Error

Approach:

1. **File Creation & Write**: The file "student.txt" is opened in write mode ("w"), and a string is written to it.
2. **Examine read(5)**: When the file is reopened in read mode, note that read(5) retrieves the first 5 characters of the file's content.
3. **Character Count**: Consider how many total characters were written and which 5 will appear first.
4. **Verify**: Conceptually slice the string to see what is returned by read(5).

Question 10

What will be the output of the following code?

```
f1 = open("student.txt", "w")
f1.write("Line 1\nLine 2\nLine 3")
f1.close()

f1 = open("student.txt", "r")
print(len(f1.readlines()))
f1.close()
```

Note: student.txt file never existed before.

Options: A) 1

B) 2

C) 3

D) Error

Approach:

1. **Check Written Content**: The code writes three separate lines ("Line 1", "Line 2", and "Line 3"), each separated by \n.
2. **Look at readlines()**: readlines() reads the entire file and returns a list where each line (up to the newline) is an element.

3. **Count the List Elements**: The output uses len(...) on that list. Think about how many lines the file now contains.
4. **Be Aware of Newlines**: Verify that each \n creates a new line in the file.

Question 11

What will be the output of the following code?

```
f1 = open("student.txt", "w")
f1.write("Python Programming")
f1.close()

f1 = open("student.txt", "r+")
f1.write("Java")
f1.seek(0)
print(f1.read())
f1.close()
```

Note: student.txt file never existed before.

Options: A) Python Programming

B) Java Programming

C) Java

D) Javaon Programming

Approach:

1. **Initial Write**: The file is first written with the text "Python Programming".
2. **r+ Mode Behavior**: Opening in read+write mode ("r+") **does not** clear the file. Instead, writing begins at the current file pointer (which starts at the beginning).
3. **Overwriting**: Writing "Java" at the start overwrites the first four characters of the existing text. Visualize the text after those characters get replaced.
4. **Seek and Read**: seek(0) moves the pointer to the file's beginning, and read() prints the entire modified content.

Question 12

What will be the output of the following code?

```
f1 = open("student.txt", "w")
f1.write("A\nB\nC\nD")
f1.close()

f1 = open("student.txt", "r")
f1.seek(2)
print(f1.read())
f1.close()
```

Note:

- Here, we are using Linux based operating system.
- student.txt file never existed before.

Options: A)

A

B

C

D

B)

B

C

D

C)

C

D

D) Error

Approach:

1. **Examine the Written Text**: The string "A\nB\nC\nD" has specific characters and newlines.
2. **Indexing Characters**: Think about the exact sequence of characters (including \n). Write down the indices:
    - Index 0 = 'A'
    - Index 1 = '\n'

- Index 2 = 'B'
- Index 3 = '\n', etc.

3. **seek(2)**: Determine which character is at position 2 and how read() will proceed from there.
4. **Resulting Substring**: After moving the pointer to index 2, the rest of the file (including newlines) is read.

Question 13

What will be the output of the following code?

```
f1 = open("student.txt", "w")
f1.write("Hello\\\\nWorld")
f1.close()

f1 = open("student.txt", "a")
f1.write("\\\\nPython")
f1.close()

f1 = open("student.txt", "r")
print(f1.readlines())
f1.close()
```

Note: student.txt file never existed before.

Options: A) ['Hello\\\\n', 'World\\\\n', 'Python']

B) ['Hello\\\\n', 'World\\\\nPython']

C) ['Hello\\\\n', 'WorldPython']

D) ['Hello\\\\nWorld\\\\nPython']

Approach:

1. **Literal Backslashes**: Notice the use of **double** backslashes. Each pair \\ represents a single backslash in the actual file content.
2. **First Write**: The initial string "Hello\\\\nWorld" ends up in the file—decide how many literal backslashes appear in "student.txt".
3. **Append**: The second write appends "\\\\nPython" at the end of the file, without overwriting.

4. **Read with readlines()**: Since the code never actually writes real newline characters (\n), think about whether the file content remains a single line or multiple lines.
5. **Resulting List**: readlines() will return a list of lines. Determine how many lines you'd see and how the backslashes are interpreted.

Question 14

What will be the output of the following code?

```
try:
   with open("nonexistent.txt", "r") as f1:
      print(f1.read())
except FileNotFoundError:
   print("File not found")
else:
   print("File read successfully")
finally:
   print("Operation complete")
```

Note: nonexistent.txt file never existed before.

Options: A)

File not found

Operation complete

B)

File read successfully

Operation complete

C)

File not found

D) Error

Approach:

1. **Expecting an Error?**: The file "nonexistent.txt" does not exist, so a FileNotFoundError is likely.
2. **Flow of try-except-else-finally**:
    - try: Attempts to open and read the file.

- except FileNotFoundError: Handles the specific exception if the file doesn't exist.
- else: Runs only if there were **no** exceptions.
- finally: Runs **regardless** of whether an exception occurred or not.

3. **Identify Which Blocks Execute**: Determine how the program flow proceeds when the file is missing.

Question 15

What will be the output of the following code?

```
f1 = open("student.txt", "w")
f1.write("Reg_no\\tName\\tMark\\n1\\tAlice\\t90\\n2\\tBob\\t85")
f1.close()

with open("student.txt", "r") as f1:
    lines = f1.readlines()
    print(lines[0].split("\\t")[3])
```

Note: student.txt file never existed before.

Options: A) Reg_no

B) Alice

C) 90

D) Bob

Approach:

1. **Writing Escaped Characters**: Notice the string uses \\t and \\n; these are **literal** backslashes in the file, not actual tab/newline characters.
2. **Inspect File Content**: The file actually stores something like "Reg_no\tName\tMark\n1\tAlice\t90\n2\tBob\t85" (with literal \t and \n).
3. **Reading and Splitting**: The code reads everything into lines[0] (only one line if no real \n is written). Then it does .split("\\t").
4. **Focus on Index [3]**: After splitting on the literal \t, figure out how many segments the list will have and which piece is at position 3.