

AI Assisted Coding Lab 13.4

Veekshitha Adharasani

2303A52175

Batch - 41

Question 1: Refactoring Data Transformation Logic

Legacy Code:

```
values = [2, 4, 6, 8, 10]
doubled = []
for v in values:
    doubled.append(v*2)
print(doubled)
```

Prompt:

Review the legacy code given that computes transformed values using an explicit loop. Suggest a more pythonic refactoring approach. Refactor the code using list comprehensions or helper functions while preventing the output. Give me a more efficient and readable code with proper inline comments, docstrings and explanations.

Code:

```
def transform_values(values, multiplier=2):
    """
    Transform a list of values by multiplying each element.
    Args:
        values (list): List of numeric values to transform
        multiplier (int): The factor to multiply each value by (default: 2)
    Returns:
        list: A new list with transformed values
    Example:
        >>> transform_values([1, 2, 3], multiplier=3)
        [3, 6, 9]
    """
    # Use list comprehension for efficient, Pythonic transformation
    return [v * multiplier for v in values]

def transform_values_conditional(values, multiplier=2, threshold=5):
    """
    Transform values with a conditional filter.
    Args:
        values (list): List of numeric values to transform
        multiplier (int): The factor to multiply each value by (default: 2)
        threshold (int): Only transform values greater than this (default: 5)
    Returns:
        list: A new list with conditionally transformed values
    """
    # Advanced list comprehension with conditional logic
    return [v * multiplier for v in values if v > threshold]

# Example usage section
if __name__ == "__main__":
    # Original data
    values = [2, 4, 6, 8, 10]
    # Pythonic approach 1: Direct list comprehension (simple and readable)
    doubled = [v * 2 for v in values]
    # Output prevented as requested
    print("Doubled: {}".format(doubled))
```

Output:

```
● PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshitha/AppData/Local/Programs/Python/Python310/python.exe d:/veekshitha/AI_Coding/Lab_13.4.py
  Doubled: [4, 8, 12, 16, 20]
○ PS D:\veekshitha\AI_Coding> []
```

Explanation:

This code defines two functions that transform lists of numbers efficiently using list comprehensions. The first function `transform_values()` multiplies each number by a given factor. The second function `transform_values_conditional()` does the same but only for numbers above a threshold. In the example section, the list `[2, 4, 6, 8, 10]` is doubled to `[4, 8, 12, 16, 20]` using a list comprehension, which is faster and more readable than traditional loops. List comprehensions are preferred in Python because they're concise, efficient, and easy to understand at a glance.

Question 2: Improving Text Processing Code Readability

Legacy Code:

```
words = ["Refactoring", "with", "AI", "improves", "quality"]
message = ""
for w in words:
    message += w + " "
print(message.strip())
```

Prompt:

Analyze the legacy code that constructs a sentence using repeated string concatenation. Suggest a more efficient and readable approach. Refactor the code accordingly while keeping the final output unchanged. Give a final efficient full code with proper docstring, inline comments and explanation.

Code:

```
def build_sentence(words):
    """
    Return a single sentence formed by joining the given words with spaces.

    Args:
        words (list[str]): sequence of words to be combined.

    Returns:
        str: the words separated by single spaces.

    Example:
        >>> build_sentence(["hello", "world"])
        'hello world'
    """
    # join is O(n) and avoids repeated string copying
    return " ".join(words)

if __name__ == "__main__":
    # original list of words
    words = ["Refactoring", "with", "AI", "improves", "quality"]

    # build the message in a single, efficient operation
    message = build_sentence(words)

    # output remains the same as before
    print(message)
```

Output:

```
● PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python310/python.exe d:/veekshitha/AI_Coding/Lab_13.4.py
Refactoring with AI improves quality
○ PS D:\veekshitha\AI_Coding> []
```

Explanation:

Instead of looping and adding each word to a string one by one which makes Python copy the whole string each time, we use str.join to glue the list of words together with spaces in a single step. That's both faster and clearer, and it still prints the same sentence.

Question 3: Safer Access to Configuration Data

Legacy Code:

```
config = {"host": "localhost", "port": 8080}
if "timeout" in config:
    print(config["timeout"])
else:
    print("Default timeout used")
```

Prompt:

Review the legacy code that manually checks for dictionary keys. Suggest to refactor the code using safer dictionary access methods. Ensure the behavior remains the same for missing keys. Give a final full efficient code with proper inline comments, docstrings and explanation.

Code:

```
def print_timeout(config, default_message="Default timeout used"):
    """
    Display the value of the 'timeout' entry in a configuration mapping.

    If the key is missing the given default message is printed instead.

    Args:
        config (dict): configuration dictionary that may contain 'timeout'.
        default_message (str): text shown when no timeout key exists.

    Returns:
        None
    """
    # dict.get returns the value if the key exists, otherwise the second
    # argument (or None if omitted); it never raises KeyError.
    timeout = config.get("timeout")
    if timeout is not None:
        # same output as original code when key is present
        print(timeout)
    else:
        # behaviour for missing key remains unchanged
        print(default_message)

if __name__ == "__main__":
    # example configuration with no timeout entry
    config = {"host": "localhost", "port": 8080}

    # helper encapsulates the lookup logic; exactly the same printed
    print_timeout(config)
```

Output:

```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python310/
● Default timeout used
○ PS D:\veekshitha\AI_Coding>
```

Explanation:

Instead of first asking “is the key there?” and then looking it up, the code calls `dict.get` which does both at once and returns a fallback if the key isn’t found. That means you never risk a `KeyError`, the logic is shorter and clearer, and the printed result is exactly the same – the timeout value if it exists, or the default message otherwise. Wrapping it in a small function makes the intent obvious and lets you reuse or change the default in one place.

Question 4: Refactoring Conditional Logic for Scalability

Legacy Code:

```
action = "divide"
x, y = 10, 2
if action == "add":
    result = x + y
elif action == "subtract":
    result = x - y
elif action == "multiply":
    result = x * y
elif action == "divide":
    result = x / y
else:
    result = None
print(result)
```

Prompt:

Examine the multiple if-elif conditions used to determine operations. Suggest a cleaner, scalable alternative. Refactor the logic using mapping techniques while preserving functionality. Give a full efficient code with proper docstring, inline comments and explanation.

Code:

```
from operator import add, sub, mul, truediv
def compute(a, b, action):
    """
    Execute a simple arithmetic operation on `a` and `b`.
    Parameters
    -----
    a : number
        First operand.
    b : number
        Second operand.
    action : str
        One of ``add``, ``subtract``, ``multiply`` or ``divide``.
    Returns
    -----
    number or None
        Result of the requested operation, or `None` if `action` is not
        recognised. Matches the behaviour of the original if/elif chain.
    """
    # dictionary mapping action names to the corresponding function
    ops = {
        "add": add,
        "subtract": sub,
        "multiply": mul,
        "divide": truediv, # true division as in original code
    }
    func = ops.get(action) # safe lookup; returns None if key is missing
    if func is None:
        # unrecognised action - same as setting result = None previously
        return None
    return func(a, b) # call the selected operator
if __name__ == "__main__":
    # example usage that mirrors the legacy code
    action = "divide"
    x, y = 10, 2
    result = compute(x, y, action)
    print(result) # prints 5.0, as before
```

Output:

```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/  
5.0  
PS D:\veekshitha\AI_Coding> []
```

Explanation:

Instead of a long if-elif-else chain that checks the action name one by one, the refactor puts each name next to the function that does it in a dictionary. Then look up the action once with .get() and call whatever you find or return None if the key isn't there. This makes the code shorter, easier to read, and simple to extend just add another entry to the map, while behaving exactly the same as before.

Question 5: Simplifying Search Logic in Collections

Legacy Code:

```
inventory = ["pen", "notebook", "eraser", "marker"]  
found = False  
for item in inventory:  
    if item == "eraser":  
        found = True  
        break  
print("Item Available" if found else "Item Not Available")
```

Prompt:

Examine the legacy code and identify the explicit loop used for searching an item. Refactor the logic into a more concise and readable form. Maintain the same output behavior. Give a full efficient code with proper inline comments, docstrings and explanation.

Code:

```
def is_item_available(inventory, item):
    """
    Return True if *item* occurs in *inventory*, False otherwise.
    The built-in membership test (`item in inventory`) iterates internally
    over the list but does so in concise, idiomatic form. The behavior of
    the legacy loop [stop when the first match is seen] is preserved.
    Parameters
    -----
    inventory : list[str]
        Sequence of product names.
    item : str
        Name to look for.
    Returns
    -----
    bool
        Availability flag.
    Examples
    -----
    >>> is_item_available(["pen", "eraser"], "eraser")
    True
    >>> is_item_available(["pen"], "marker")
    False
    """
    # simple membership test replaces the explicit loop and flag
    return item in inventory

if __name__ == "__main__":
    # sample inventory list
    inventory = ["pen", "notebook", "eraser", "marker"]

    # check availability in one expression
    available = is_item_available(inventory, "eraser")

    # same output as before
    print("Item Available" if available else "Item Not Available")
```

Output:

- PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python, Item Available
- PS D:\veekshitha\AI_Coding> []

Explanation:

The module defines a small helper that simply tests whether a given name is in the list by using Python's `in` operator, which does the same job as the old loop but in one neat expression. When run as a script it builds a sample inventory, calls that function for "eraser", and then prints "Item Available" or "Item Not Available" depending on the Boolean result. No explicit loop or flag variable is needed the membership test handles the search and short circuits as soon as a match is found.