ASSIGNMENT 13

RAMIDI . SAI CHARAN

2403A52124

TASK 1 :

```
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
    squares.append(n ** 2)
print(squares)
```

```
[1, 4, 9, 16, 25]
```

```
numbers = [1, 2, 3, 4, 5]
squares = [n ** 2 for n in numbers]
print(squares)
```

```
[1, 4, 9, 16, 25]
```

EXPLANATION :

The refactored code uses a **list comprehension** (`squares = [n ** 2 for n in numbers]`) to concisely compute the squares. This approach is **declarative**, stating *what* the list should contain, not *how* to build it step-by-step. List comprehensions are highly **Pythonic**, improving **readability** by condensing the three-line loop/append pattern into one line. They are also generally more **efficient** because Python optimizes their execution and memory allocation. This refactoring demonstrates a cleaner, more idiomatic solution for list transformation.

TASK 2 :

```
words = ["AI", "helps", "in", "refactoring", "code"]
sentence = ""
for word in words:
    sentence += word + " "
print(sentence.strip())
```

```
AI helps in refactoring code
```

```
words = ["AI", "helps", "in", "refactoring", "code"]
sentence = " ".join(words)
print(sentence)
```

```
AI helps in refactoring code
```

ASSIGNMENT 13

EXPLANATION :

This refactored code uses the `"".join()` method, which is a more efficient and Pythonic way to concatenate strings from a list.

- `" ".join(words)` takes the list of strings `words` and joins them together with a space (`" "`) in between each element.
- The result is a single string stored in the `sentence` variable.

This approach avoids the repeated creation of new strings that happens with the `+=` operator in a loop, making it more performant, especially for larger lists.

This refactored code uses a list comprehension, which is a concise way to create lists in Python.

- `[n ** 2 for n in numbers]` iterates through each element `n` in the `numbers` list and calculates its square (`n ** 2`).
- The results are collected into a new list called `squares`.

This approach is more Pythonic and often more readable than using a traditional `for` loop with `append`.

TASK 3 :

```
student_scores = {"Alice": 85, "Bob": 90}
if "Charlie" in student_scores:
    print(student_scores["Charlie"])
else:
    print("Not Found")
```

```
Not Found
```

```
student_scores = {"Alice": 85, "Bob": 90}
print(student_scores.get("Charlie", "Not Found"))
```

```
Not Found
```

EXPLANATION:

This refactored code uses the `.get()` method for dictionary lookup, which is a safer and more Pythonic way to access dictionary values.

- `student_scores.get("Charlie", "Not Found")` attempts to retrieve the value associated with the key `"Charlie"` from the `student_scores` dictionary.
- If the key `"Charlie"` is found, its corresponding value (if any) is returned.
- If the key is not found, the default value `"Not Found"` is returned instead of raising a `KeyError`.

This approach is more concise and handles missing keys gracefully without the need for an explicit `if...else` block.

This refactored code uses the `"".join()` method, which is a more efficient and Pythonic way to concatenate strings from a list.

- `" ".join(words)` takes the list of strings `words` and joins them together with a space (`" "`) in between each element.
- The result is a single string stored in the `sentence` variable.

This approach avoids the repeated creation of new strings that happens with the `+=` operator in a loop, making it more performant, especially for larger lists.

This refactored code uses a list comprehension, which is a concise way to create lists in Python.

- `[n ** 2 for n in numbers]` iterates through each element `n` in the `numbers` list and calculates its square (`n ** 2`).
- The results are collected into a new list called `squares`.

This approach is more Pythonic and often more readable than using a traditional `for` loop with `append`.

ASSIGNMENT 13

TASK 4 :

```
operation = "multiply"
a, b = 5, 3

operations = {
    "add": lambda x, y: x + y,
    "subtract": lambda x, y: x - y,
    "multiply": lambda x, y: x * y,
}

result = operations.get(operation, lambda x, y: None)(a, b)

print(result)
```

```
15
```

```
operation = "multiply"
a, b = 5, 3

if operation == "add":
    result = a + b
elif operation == "subtract":
    result = a - b
elif operation == "multiply":
    result = a * b
else:
    result = None
print(result)
```

```
15
```

EXPLANATION :

This refactored code uses the `.get()` method for dictionary lookup, which is a safer and more Pythonic way to access dictionary values.

- `student_scores.get("Charlie", "Not Found")` attempts to retrieve the value associated with the key `"Charlie"` from the `student_scores` dictionary.
- If the key `"Charlie"` is found, its corresponding value (if any) is returned.
- If the key is not found, the default value `"Not Found"` is returned instead of raising a `KeyError`.

This approach is more concise and handles missing keys gracefully without the need for an explicit `if...else` block.

This refactored code uses the `"".join()` method, which is a more efficient and Pythonic way to concatenate strings from a list.

- `" ".join(words)` takes the list of strings `words` and joins them together with a space (`" "`) in between each element.
- The result is a single string stored in the `sentence` variable.

This approach avoids the repeated creation of new strings that happens with the `+=` operator in a loop, making it more performant, especially for larger lists.

This refactored code uses a list comprehension, which is a concise way to create lists in Python.

- `[n ** 2 for n in numbers]` iterates through each element `n` in the `numbers` list and calculates its square (`n ** 2`).
- The results are collected into a new list called `squares`.

This approach is more Pythonic and often more readable than using a traditional `for` loop with `append`.

ASSIGNMENT 13

TASK 5 :

```
items = [10, 20, 30, 40, 50]
found = False
for i in items:
    if i == 30:
        found = True
        break
print("Found" if found else "Not Found")
```

```
Found
```

```
items = [10, 20, 30, 40, 50]
target = 30

if target in items:
    print("Found")
else:
    print("Not Found")
```

```
Found
```

EXPLANATION :

This refactored code uses the `in` keyword, which is a concise and Pythonic way to check for the presence of an element in a sequence (like a list, tuple, or string).

- `if target in items:` directly checks if the value of `target` exists within the `items` list.
- If the element is found, the condition is `True`, and the code inside the `if` block is executed.
- If the element is not found, the condition is `False`, and the code inside the `else` block is executed.

This approach is much simpler and more readable than using a `for` loop with a boolean flag and a `break` statement.