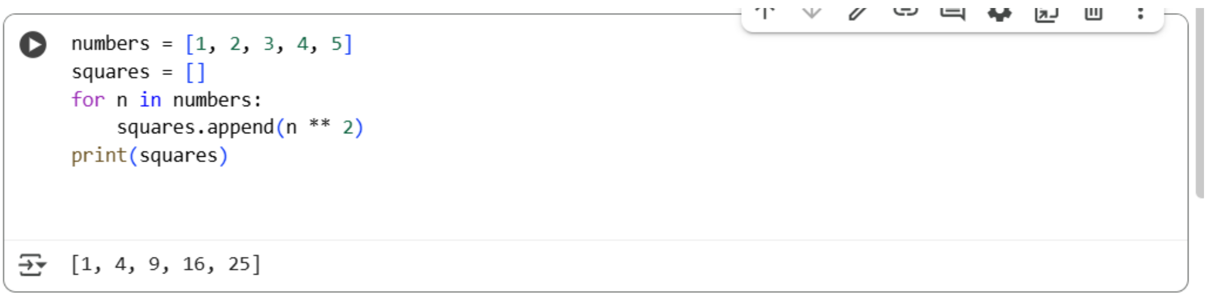
Assignment-13.4

Task 1:

Write a python program to refactor repeated loops into a cleaner, more Pythonic approach.

Code and output:

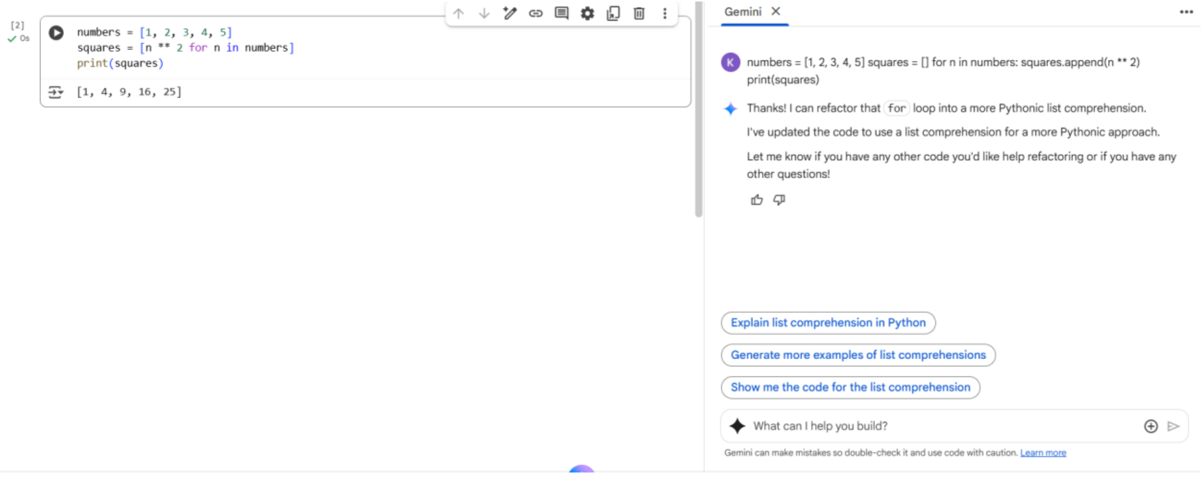
Lagacy code:



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

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

Re-generated code:



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

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

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

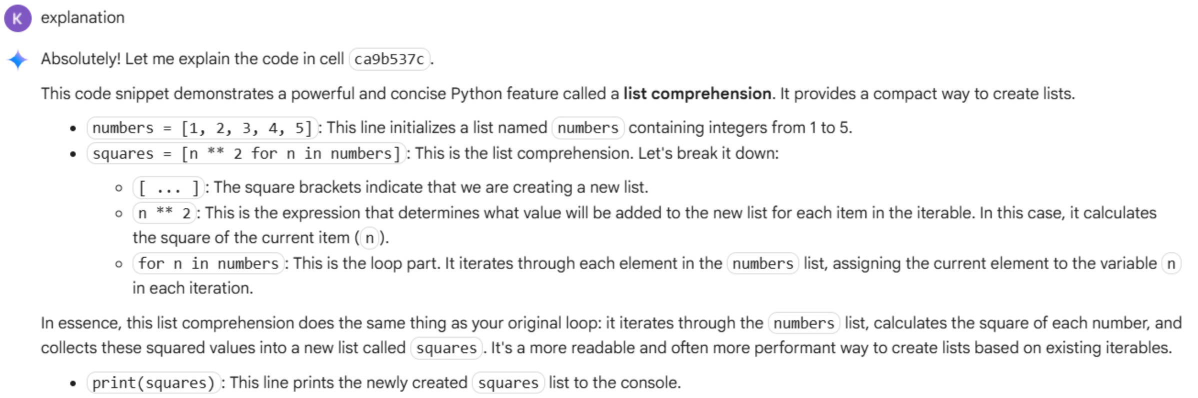
Thanks! I can refactor that `for` loop into a more Pythonic list comprehension.

I've updated the code to use a list comprehension for a more Pythonic approach.

Let me know if you have any other code you'd like help refactoring or if you have any other questions!

Explain list comprehension in Python
Generate more examples of list comprehensions
Show me the code for the list comprehension
What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



explanation

Absolutely! Let me explain the code in cell `ca9b537c`.

This code snippet demonstrates a powerful and concise Python feature called a **list comprehension**. It provides a compact way to create lists.

- `numbers = [1, 2, 3, 4, 5]`: This line initializes a list named `numbers` containing integers from 1 to 5.
- `squares = [n ** 2 for n in numbers]`: This is the list comprehension. Let's break it down:
  - `[ ... ]`: The square brackets indicate that we are creating a new list.
  - `n ** 2`: This is the expression that determines what value will be added to the new list for each item in the iterable. In this case, it calculates the square of the current item (`n`).
  - `for n in numbers`: This is the loop part. It iterates through each element in the `numbers` list, assigning the current element to the variable `n` in each iteration.

In essence, this list comprehension does the same thing as your original loop: it iterates through the `numbers` list, calculates the square of each number, and collects these squared values into a new list called `squares`. It's a more readable and often more performant way to create lists based on existing iterables.

- `print(squares)`: This line prints the newly created `squares` list to the console.

Task 2:

Write a python program to Simplify string concatenation

words = ["AI", "helps", "in", "refactoring", "code"]

sentence = ""

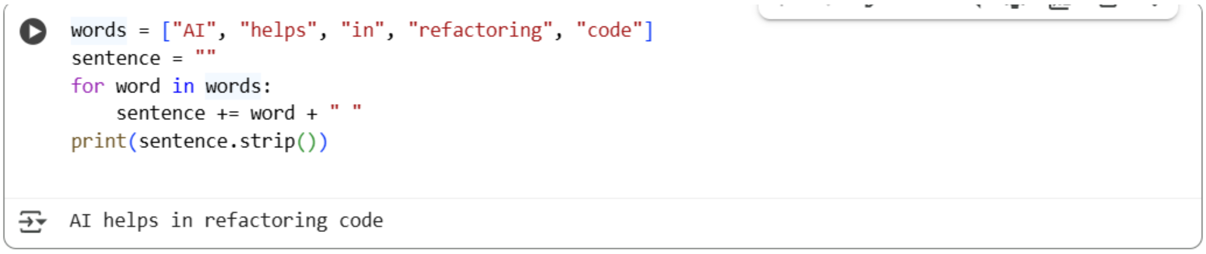
for word in words:

    sentence += word + " "

print(sentence.strip())

Code and Output:

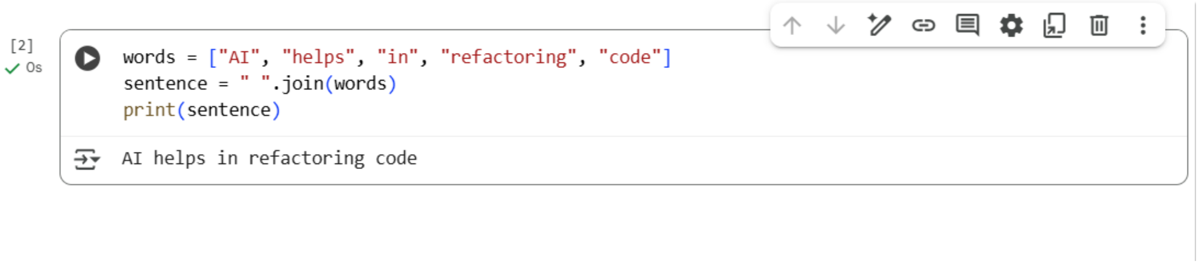
Lagecy code:



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

```
AI helps in refactoring code
```

Re-generated code:

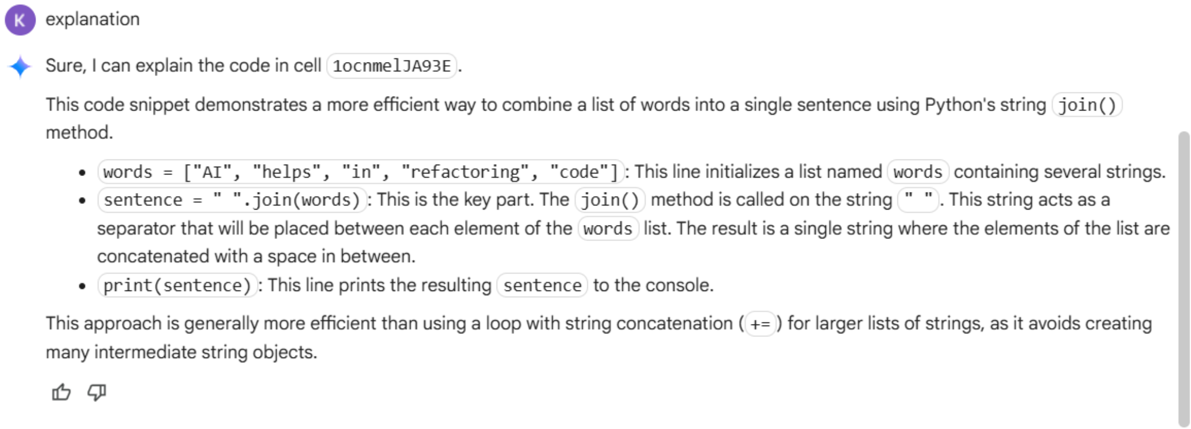


[2]
✓ 0s

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

```
AI helps in refactoring code
```

Explanation:



K explanation

Sure, I can explain the code in cell `1ocnmelJA93E`.

This code snippet demonstrates a more efficient way to combine a list of words into a single sentence using Python's string `join()` method.

- `words = ["AI", "helps", "in", "refactoring", "code"]`: This line initializes a list named `words` containing several strings.
- `sentence = " ".join(words)`: This is the key part. The `join()` method is called on the string `" "`. This string acts as a separator that will be placed between each element of the `words` list. The result is a single string where the elements of the list are concatenated with a space in between.
- `print(sentence)`: This line prints the resulting `sentence` to the console.

This approach is generally more efficient than using a loop with string concatenation (`+=`) for larger lists of strings, as it avoids creating many intermediate string objects.

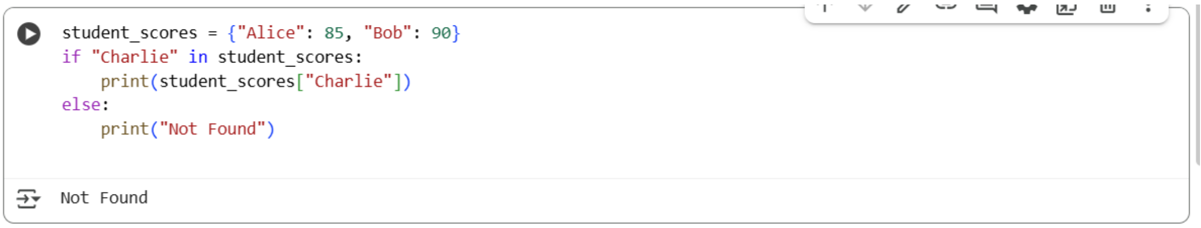
Task 3:

Write a python program to replace manual dictionary lookup with a safer method.

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

Code and output:

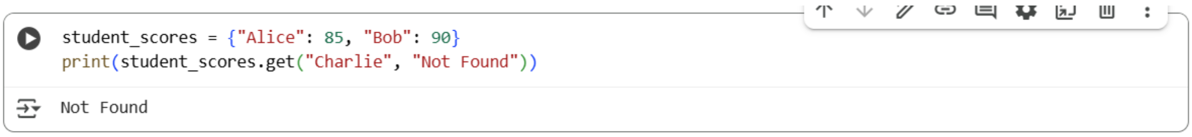
Lagecy code:



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

```
Not Found
```

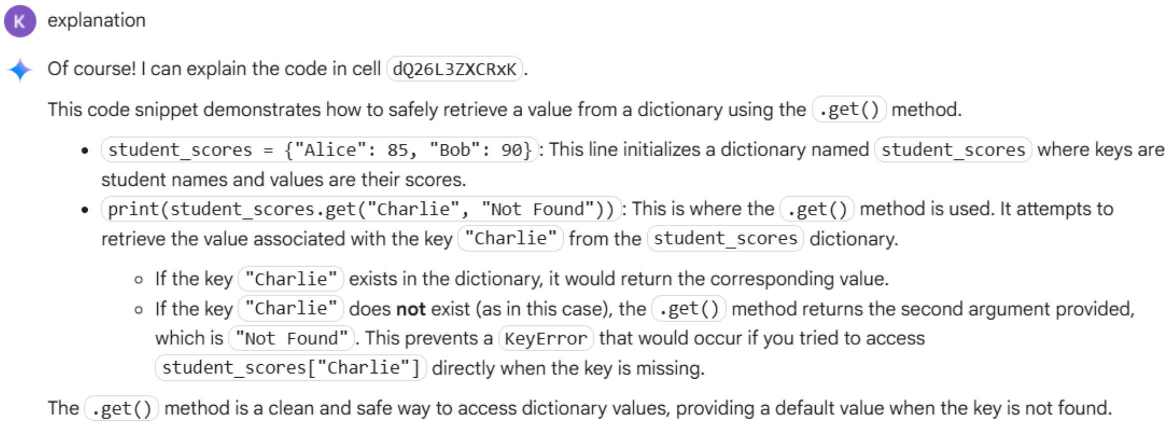
Re-generated code:



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

```
Not Found
```

Explanation:



explanation

Of course! I can explain the code in cell `dQ26L3ZXCRxK`.

This code snippet demonstrates how to safely retrieve a value from a dictionary using the `.get()` method.

- `student_scores = {"Alice": 85, "Bob": 90}`: This line initializes a dictionary named `student_scores` where keys are student names and values are their scores.
- `print(student_scores.get("Charlie", "Not Found"))`: This is where the `.get()` method is used. It attempts to retrieve the value associated with the key `"Charlie"` from the `student_scores` dictionary.
    - If the key `"Charlie"` exists in the dictionary, it would return the corresponding value.
    - If the key `"Charlie"` does **not** exist (as in this case), the `.get()` method returns the second argument provided, which is `"Not Found"`. This prevents a `KeyError` that would occur if you tried to access `student_scores["Charlie"]` directly when the key is missing.

The `.get()` method is a clean and safe way to access dictionary values, providing a default value when the key is not found.

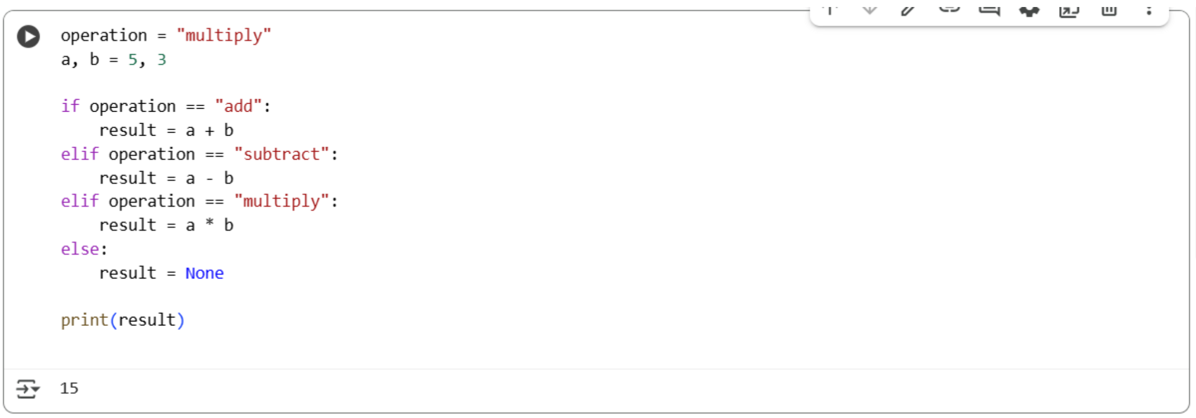
Task 4:

Write a python program to refactor repetitive if-else blocks

```
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)
```

code and output:

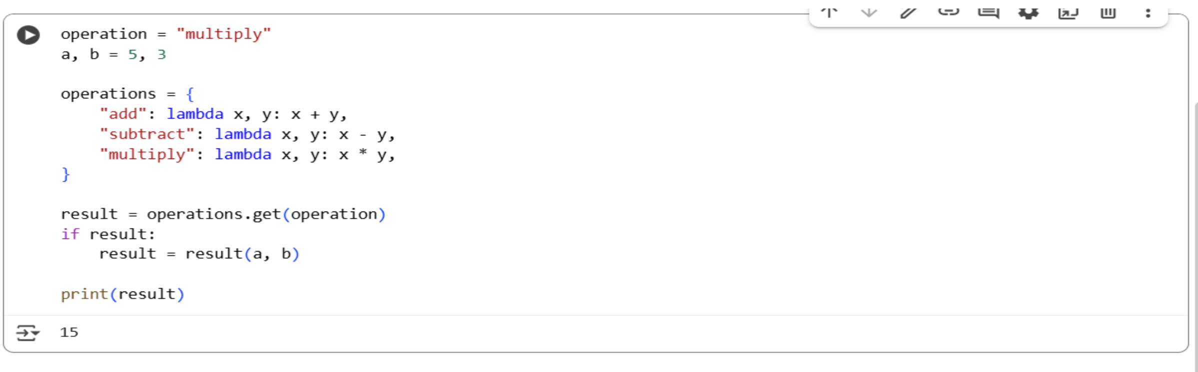
Lagecy code:



```
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
```

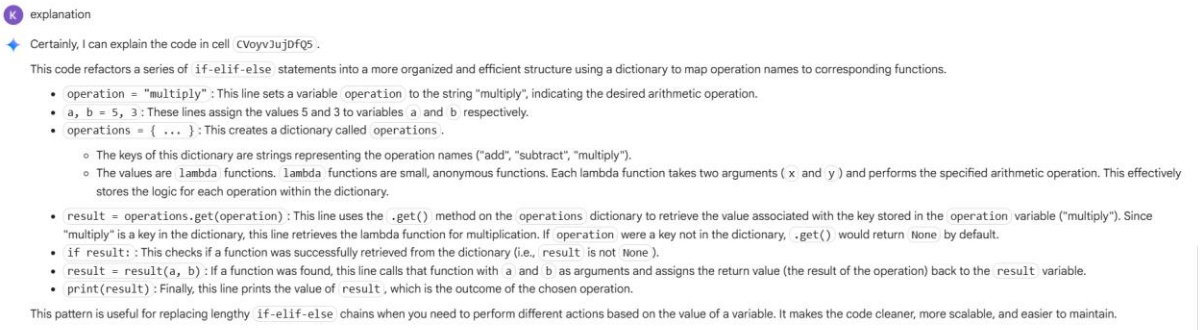
Re-generated code:



```
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)
if result:
    result = result(a, b)

print(result)
```

```
15
```

Explanation:



explanation

Certainly, I can explain the code in cell `CVoyvJujDfQ5` .

This code refactors a series of `if-elif-else` statements into a more organized and efficient structure using a dictionary to map operation names to corresponding functions.

- `operation = "multiply"` : This line sets a variable `operation` to the string "multiply", indicating the desired arithmetic operation.
- `a, b = 5, 3` : These lines assign the values 5 and 3 to variables `a` and `b` respectively.
- `operations = { ... }` : This creates a dictionary called `operations` .
  - The keys of this dictionary are strings representing the operation names ("add", "subtract", "multiply").
  - The values are `lambda` functions. `lambda` functions are small, anonymous functions. Each lambda function takes two arguments (`x` and `y`) and performs the specified arithmetic operation. This effectively stores the logic for each operation within the dictionary.
- `result = operations.get(operation)` : This line uses the `.get()` method on the `operations` dictionary to retrieve the value associated with the key stored in the `operation` variable ("multiply"). Since "multiply" is a key in the dictionary, this line retrieves the lambda function for multiplication. If `operation` were a key not in the dictionary, `.get()` would return `None` by default.
- `if result:` : This checks if a function was successfully retrieved from the dictionary (i.e., `result` is not `None`).
- `result = result(a, b)` : If a function was found, this line calls that function with `a` and `b` as arguments and assigns the return value (the result of the operation) back to the `result` variable.
- `print(result)` : Finally, this line prints the value of `result` , which is the outcome of the chosen operation.

This pattern is useful for replacing lengthy `if-elif-else` chains when you need to perform different actions based on the value of a variable. It makes the code cleaner, more scalable, and easier to maintain.

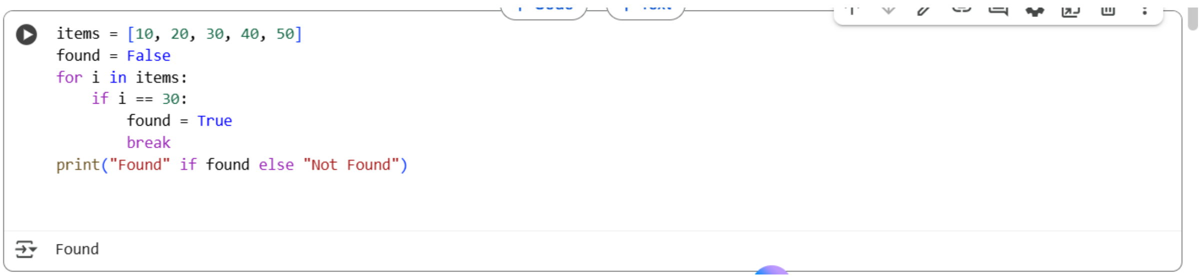
Task 5:

Write a python program to Optimize nested loops for searching.

```
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")
```

Code and output:

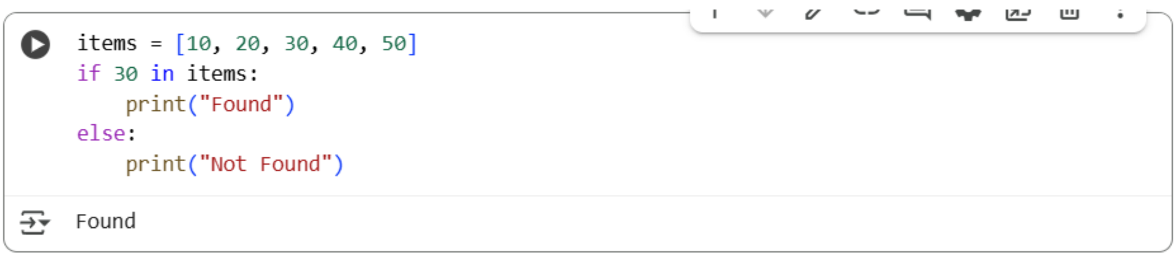
Lagecy_code:



```
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
```

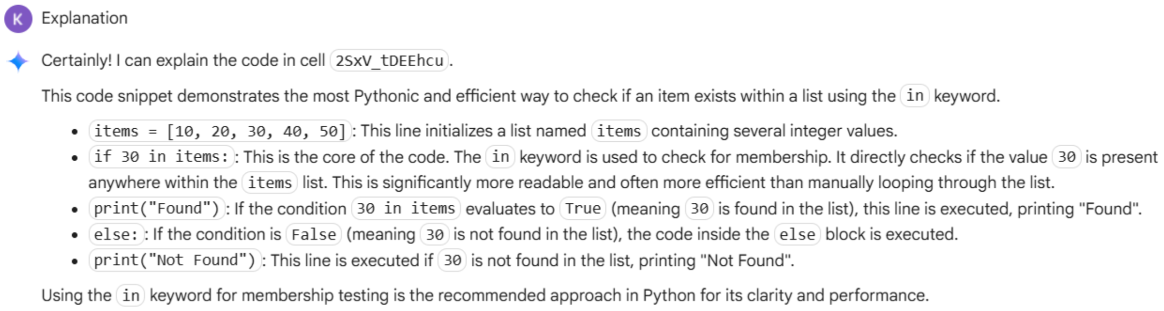
Re-generated code:



```
items = [10, 20, 30, 40, 50]
if 30 in items:
    print("Found")
else:
    print("Not Found")
```

```
Found
```

Explanation:



Explanation

Certainly! I can explain the code in cell `2SxV_tDEEhcu`.

This code snippet demonstrates the most Pythonic and efficient way to check if an item exists within a list using the `in` keyword.

- `items = [10, 20, 30, 40, 50]`: This line initializes a list named `items` containing several integer values.
- `if 30 in items:`: This is the core of the code. The `in` keyword is used to check for membership. It directly checks if the value `30` is present anywhere within the `items` list. This is significantly more readable and often more efficient than manually looping through the list.
- `print("Found")`: If the condition `30 in items` evaluates to `True` (meaning `30` is found in the list), this line is executed, printing "Found".
- `else:`: If the condition is `False` (meaning `30` is not found in the list), the code inside the `else` block is executed.
- `print("Not Found")`: This line is executed if `30` is not found in the list, printing "Not Found".

Using the `in` keyword for membership testing is the recommended approach in Python for its clarity and performance.