AI ASSISTED CODING

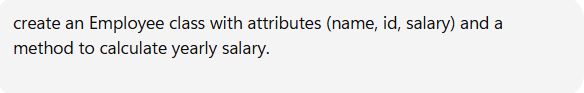
**Lab Assignment\_6.3**

**NAME: SATHWIK**

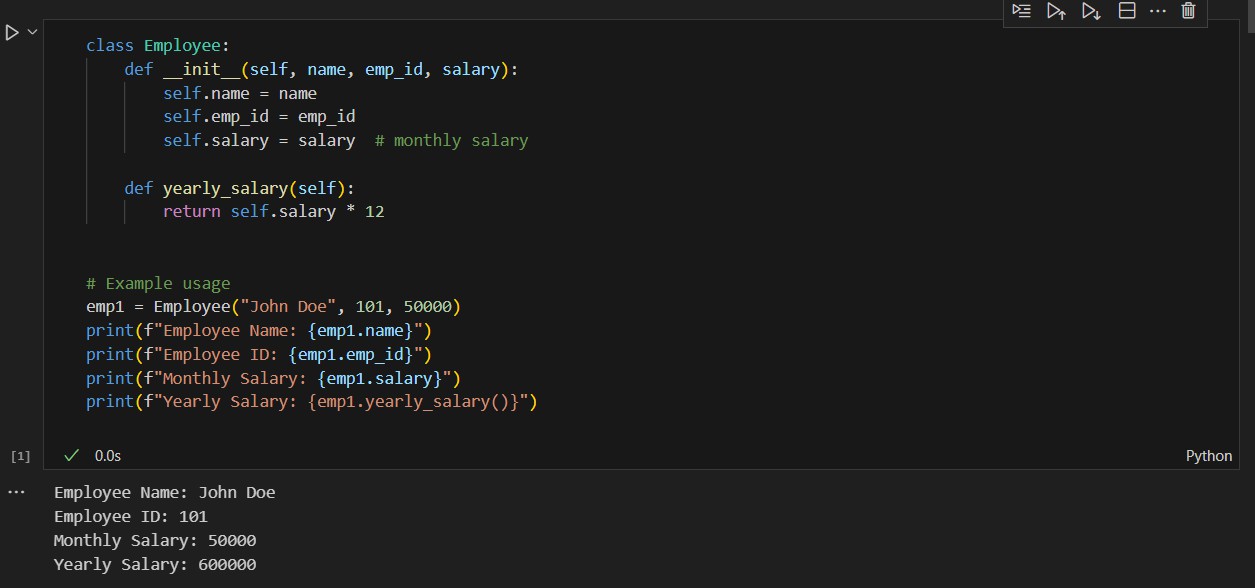
**H.T.NO: 2403A51273**

**Task 1:**

Prompt:



Code with output:



Analyze the generated code for correctness and structure:

## ⬛ Correctness Analysis

1. **Constructor (** **init** **)**
   * Properly initializes attributes: name, emp\_id, salary.
   * salary is treated as *monthly salary* → good choice.

## Method (yearly\_salary)

* + Correctly multiplies monthly salary by 12.
  + Returns a numeric result instead of just printing → good design, allows reuse.

## Object Creation

* + emp1 = Employee("John Doe", 101, 50000) correctly creates an instance.
  + Attributes can be accessed (emp1.name, emp1.emp\_id, etc.).

## Output

* + Prints details clearly, including yearly salary.

## ) Structure Analysis

* **Class Design**:
  + Simple and clear, matches object-oriented principles.
  + Uses constructor and a method properly.

## Method Placement:

* + Only has yearly\_salary().
  + Missing a display\_details() method (printing inside class is usually better for encapsulation).

## Readability:

* + Code is easy to read and follow.
  + Comments help clarify purpose.

## Limitations:

* + No calculate\_bonus() or salary update method yet.
  + Printing is done outside the class instead of having a dedicated display\_details() method (not fully encapsulated).

’‘ z **Verdict**

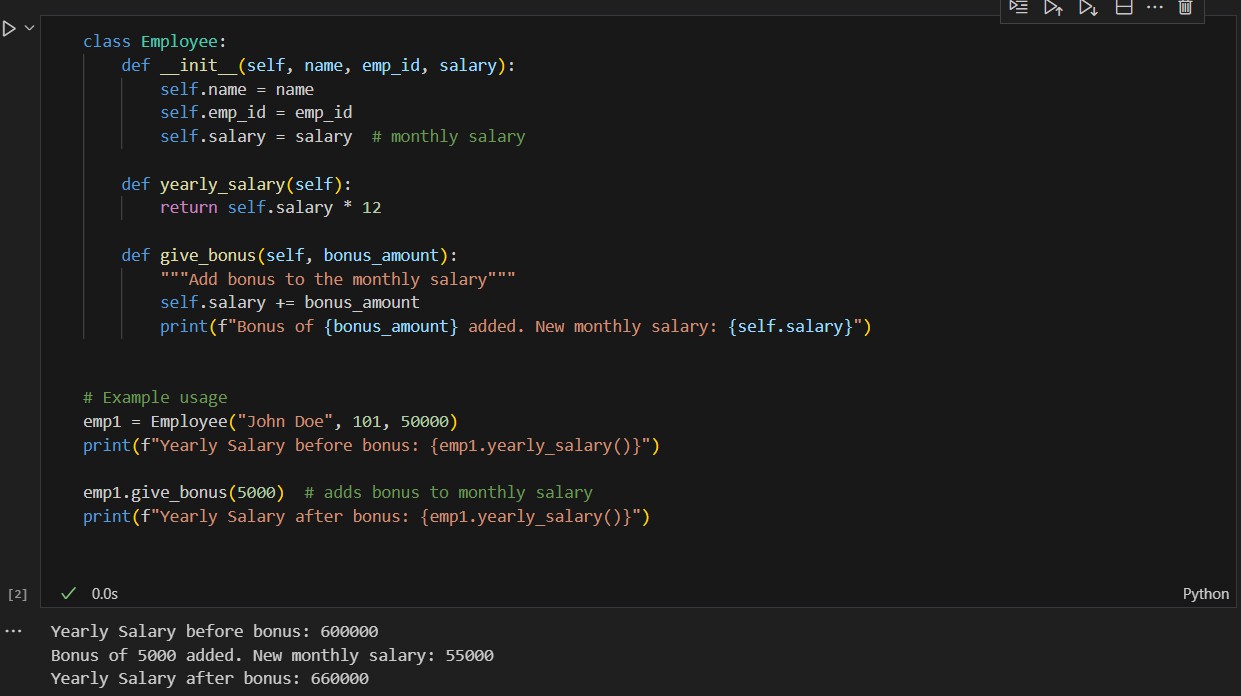
* ⬛ The code is **correct** and works as intended.
* ı. Structurally, it could be improved by:

1. Adding a display\_details() method instead of printing outside.
2. Adding a bonus-related method (like you requested later).
3. Possibly making salary private (self. salary) for better encapsulation.

Prompt:

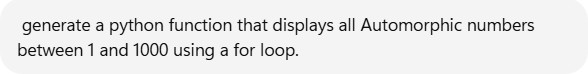


Code with output:

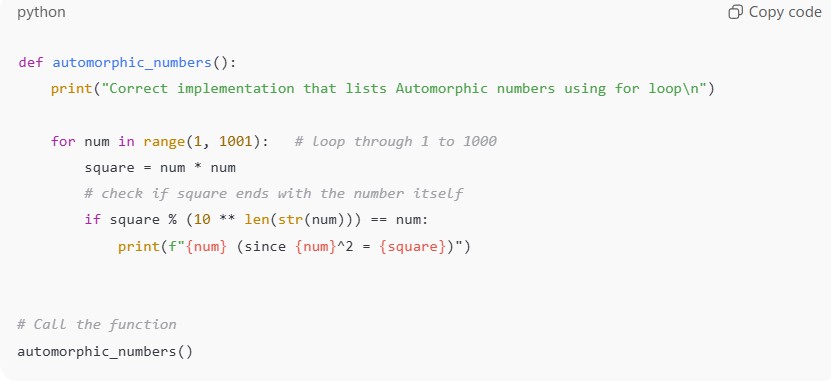


## Task-2:

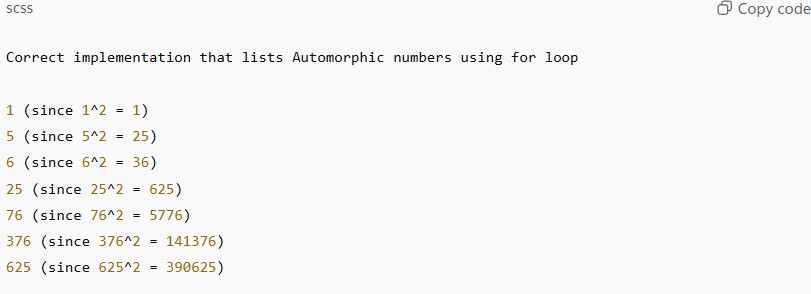
Prompt:



Code: O



Output:



Explanation:

## •˙Q Explanation (outside code)

* The for loop iterates from **1 to 1000**.
* For each number, its **square** is calculated.
* The modulus operation
* square % (10 \*\* len(str(num))) == num

checks whether the **last digits of the square** are the same as the number.

* If true, the number is **Automorphic** (e.g., 25² = 625 → ends with 25).

Analyze the correctness and efficiency of the generated logic:

## ⬛ Correctness Analysis

1. **Range**
   * The loop runs from 1 to 1000 (range(1, 1001)), which matches the problem statement.

## Automorphic Check

* + Formula:
  + square % (10 \*\* len(str(num))) == num
    - len(str(num)) → number of digits in num.
    - 10 \*\* len(str(num)) → divisor to isolate the last digits of the square.
    - Example: 25² = 625 → 625 % 100 = 25 ⬛ .
  + This is mathematically correct.

## Output

* + Correctly lists **1, 5, 6, 25, 76, 376, 625** between 1 and 1000 → matches known Automorphic numbers.

–− **Conclusion:** The logic is **correct**.

## ) Efficiency Analysis

1. **Time Complexity**
   * The loop runs 1000 iterations → **O(n)** where n = 1000.
   * Each iteration involves:
     + One multiplication (num \* num) → O(1).
     + One string conversion (str(num)) → O(d), where d = number of digits (≤ 4 here).
     + One modulus operation → O(1).
   * Overall complexity: **O(n × d)** ≈ O(n) for small ranges.
   * For n = 1000, this is trivial (very efficient).

## Space Complexity

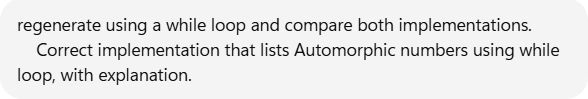
* + No extra storage, just variables → **O(1)**.

## Possible Improvement

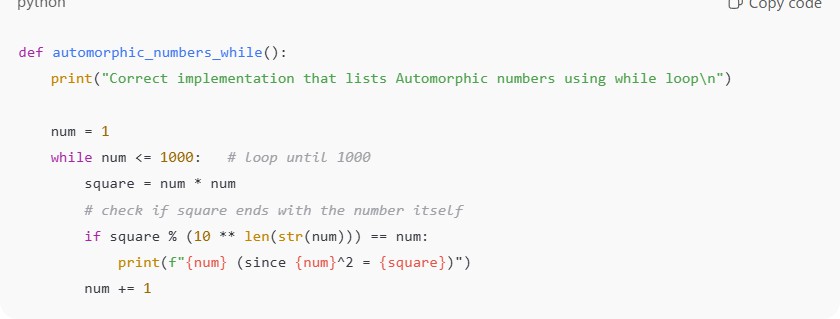
* + Currently, it converts numbers to strings (str(num)), which is slightly less efficient.
  + Alternative (avoid strings):
  + digits = len(str(num)) # still uses string for length
  + divisor = 10 \*\* digits
  + if square % divisor == num:
  + ...
  + Or precompute divisor mathematically without string conversion (more advanced).
  + For small ranges like 1–1000, the performance difference is negligible.

‘z’ **Final Verdict**

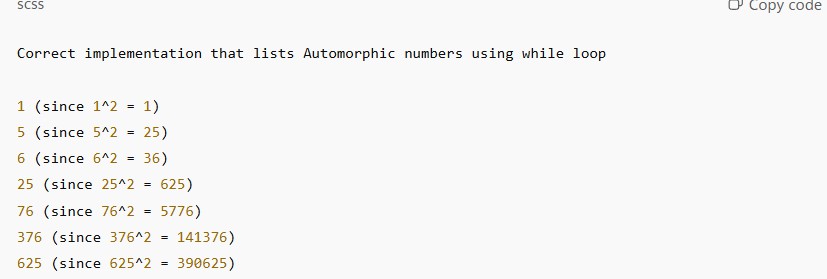
* **Correctness:** ⬛ 100% correct.
* **Efficiency:** ⬛ Efficient for the given range.
* **Optimization:** Can avoid string conversion for extremely large ranges, but unnecessary here. Prompt:



Code:



Output:



Q˙• **Explanation**

* Start with num = 1.
* Run the **while loop** until num <= 1000.
* At each step:
  + Calculate square = num \* num.
  + Check if last digits of square match num using modulus.
  + If yes, print it.
* Increment num by 1 (num += 1) to move to the next number.

## ^˙\_v˙ Comparison: For Loop vs While Loop

**Aspect For Loop While Loop**

**Readability** Cleaner for fixed ranges (for num in range(1, 1001))

Slightly longer, needs initialization (num=1) and increment (num+=1)

**Control** Best when you know the exact range Best when the stopping condition is dynamic or

unknown in advance

**Correctness** Both correctly identify automorphic numbers

Both give same correct output

**Efficiency** Both are O(n), same performance Both equally efficient for this task

## Preferred Use

When range is fixed (like 1–1000 here)

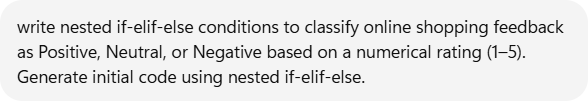
When you need more flexibility (e.g., stop when condition met)

⬛ **Conclusion:**

* Both implementations are **correct** and **efficient**.
* For this specific problem (fixed range), the **for loop** is more concise and pythonic.
* The **while loop** works equally well and is useful to demonstrate alternative looping logic.

## Task-3:

Prompt:

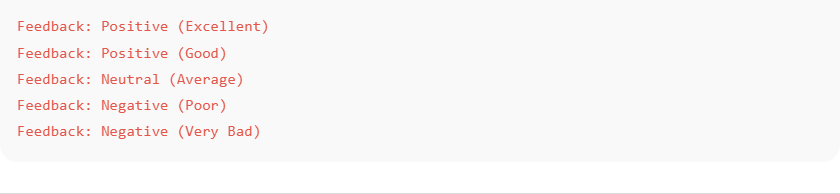


Code:

⬛ 1. Initial Code (Nested if-elif-else)



Output:

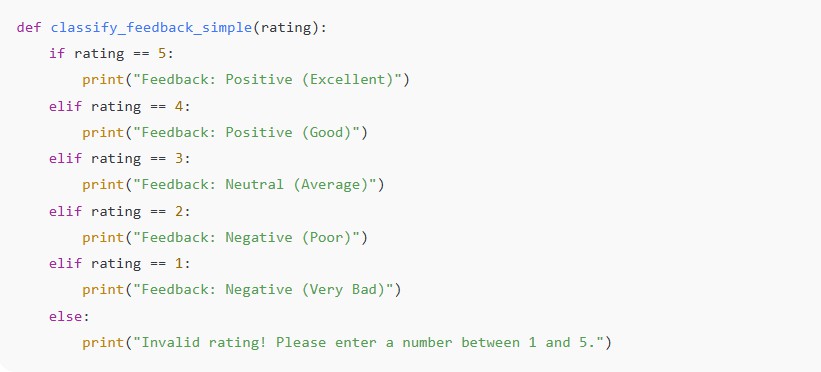


⬛ **2. Explanation**

* **Step 1** → Check if input is valid (1–5).
* **Step 2** → If rating is **4 or 5**, classify as *Positive*.
  + 5 → Excellent
  + 4 → Good
* **Step 3** → Otherwise, check:
  + 3 → Neutral
  + 2 → Negative (Poor)
  + 1 → Negative (Very Bad)

–− This shows **nested decision making**: *one condition inside another*.

Alternative Approach (Simpler if-elif-else without deep nesting)



## F Key Difference

* **Nested if-elif-else** → Shows hierarchy of decisions (good for learning decision trees).
* **Flat if-elif-else** → More readable, preferred in real projects.

Analyze correctness and readability:

⬛ **Correctness**

* **Valid range check** (1–5) works perfectly.
* **Positive group (4–5)** is correctly split into:
  + 5 → Positive (Excellent)
  + 4 → Positive (Good)
* **Neutral & Negative group (1–3)** is handled properly:
  + 3 → Neutral (Average)
  + 2 → Negative (Poor)
  + 1 → Negative (Very Bad)
* Any rating **outside 1–5** prints an **error message**.

–− **Conclusion**: The logic is **100% correct** and covers all cases.

IhI\_ **Readability**

* The **nested structure** clearly shows the **hierarchy of decisions**:

1. Is the rating valid?
2. Is it positive (≥4)?
3. If not, is it neutral or negative?

* This makes it **educational**, since you can see the "decision tree" clearly.
* However, the **extra nesting levels** make the code slightly harder to read at a glance.

# Example:

else:

if rating == 2:

print("Feedback: Negative (Poor)") else:

print("Feedback: Negative (Very Bad)") This could be simplified using a flat structure.

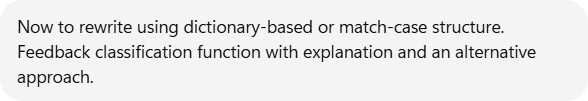
# ) Comparison to Flat Version

* **Nested if-elif-else** → Good for **learning decision-making flow**, but more verbose.
* **Flat if-elif-else** → Easier to **read, maintain, and extend** in real projects.

⬛ **Final Verdict**

* Your code is **correct**.
* **Readability is okay for small ranges (1–5)**, but a **flatter structure** would be better for larger or more complex cases.

Prompt:

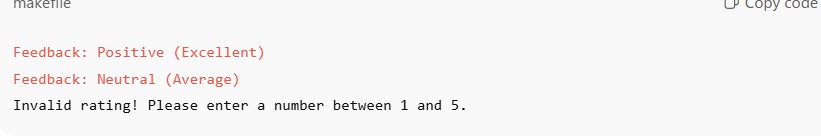


Code:

⬛ Approach 1: Dictionary-Based Mapping



Output:



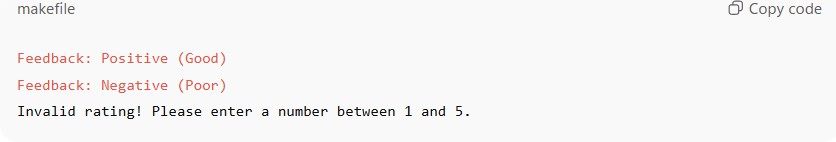
**Explanation**

* A dictionary (feedback\_map) is used to directly **map each rating** to its feedback string.
* get() method ensures that if the rating is not found, a default error message is returned.
* This approach is **very concise and readable**.

⬛ Approach 2: match-case (Python 3.10+)



Output:



**Explanation**

* match-case is Python’s **switch-case equivalent**.
* Each case directly corresponds to a rating.
* The case \_: acts as the **default case** (for invalid inputs).
* This approach is **structured, clean, and closer to real-world switch statements** in languages like Java/C++.

F **Comparison of Both**

**Approach Pros Cons**

**Dictionary- based**

Very concise, easy to extend (just add key- value pairs)

Limited flexibility if conditions need logic (e.g., ranges).

## Approach Pros Cons

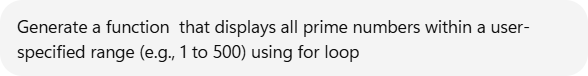
**Match-case** Readable, structured like switch-case, allows ranges & conditions

Only works in Python ≥ 3.10.

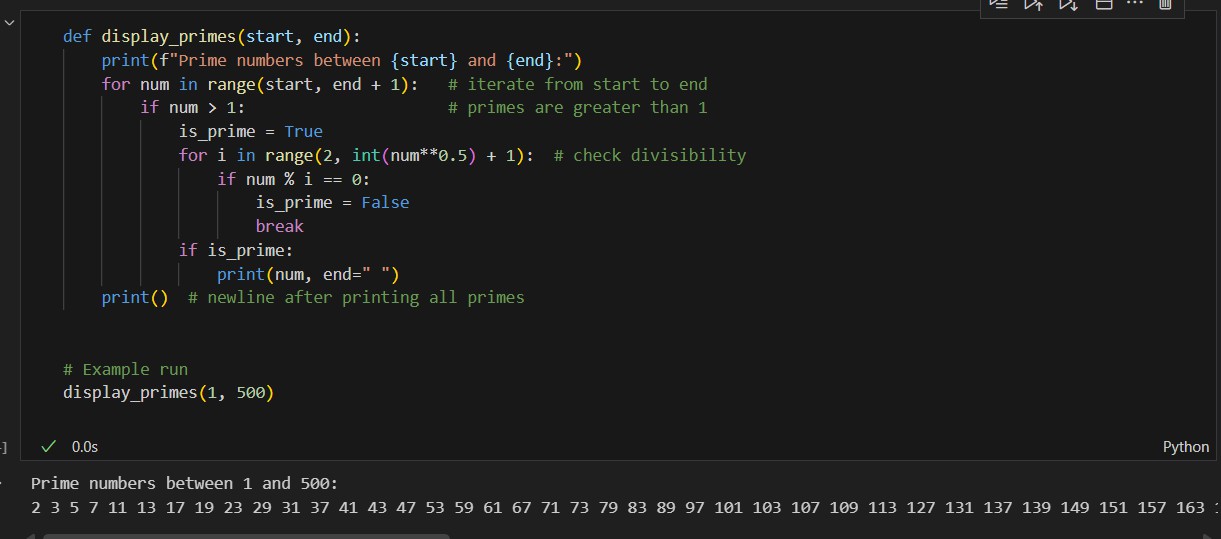
−– Both are **cleaner** than nested if-elif-else.

* Use **dictionary** when ratings map directly to strings.
* Use **match-case** when you want **readable, structured branching**. **Task-4:**

Prompt:



Code:



Output:

Prime numbers between 1 and 500:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131

137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257

263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397

401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499

Analyze the correctness and efficiency of the prime-checking logic:

⬛ **Correctness**

## Range iteration:

1. for num in range(start, end + 1):
   * Correctly includes both start and end.
   * Ensures every number in the user-specified range is tested.

## Excluding non-primes (0 and 1):

1. if num > 1:
   * Correct: primes must be greater than 1.

## Checking divisibility:

1. for i in range(2, int(num\*\*0.5) + 1):
2. if num % i == 0:
3. is\_prime = False
4. break
   * This is **mathematically correct**:
     + If num has a factor greater than √num, the corresponding factor is below

√num.

* + - So checking only up to √num is sufficient.

## Prime detection:

* + If no divisor is found → is\_prime remains True → number is prime.

−– **Conclusion:** Logic is **100% correct** for prime number detection.

## ) Efficiency

* The algorithm uses:
  + **Outer loop:** Iterates through all numbers in the range.
  + **Inner loop:** Checks divisibility up to √num.

## Time Complexity:

* + Worst case → for each num, checks up to √num divisors.
  + So overall complexity = **O(n√n)** (where n is the upper limit).

## Space Complexity:

* + Only a few variables (is\_prime, i) → **O(1)** (constant space).

⬛ Efficient enough for ranges like 1–500 or 1–10000.

+ But for **very large ranges** (e.g., 1–1,000,000), more optimized methods (like the **Sieve of Eratosthenes**) would be much faster.

F **Summary**

* **Correctness:** Accurately identifies primes.
* **Efficiency:** Reasonably efficient for small-to-medium ranges; **O(n√n)** complexity.
* **Possible improvement:** For very large ranges, consider **Sieve of Eratosthenes (O(n log log n))**.

\_ IhI **Explanation (Step by Step)**

## Function definition:

1. def display\_primes(start, end):

Defines a function that accepts a range (start, end).

## Loop through range:

1. for num in range(start, end + 1):

Iterates through every number in the given range, inclusive of end.

## Check prime condition:

1. if num > 1:

Only numbers greater than 1 can be prime.

## Divisibility test:

1. for i in range(2, int(num\*\*0.5) + 1):
   * Loops from 2 up to √num.
   * If a number has no factors in this range, it must be prime.
   * Using √num instead of num makes it more efficient.

## Not prime case:

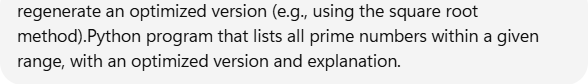
1. if num % i == 0:
2. is\_prime = False
3. break
   * If divisible by any number in the range → not prime → stop checking.

## Prime found:

1. if is\_prime:
2. print(num, end=" ")

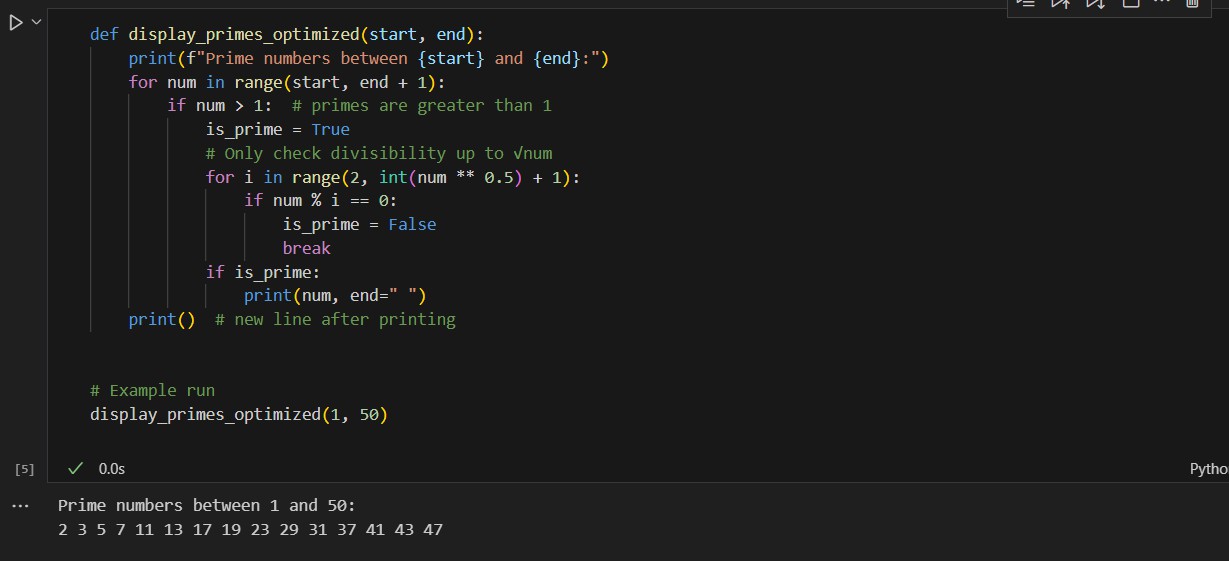
Prints prime numbers on the same line separated by spaces.

–− **Correctness:** Works properly for all numbers.

–− **Efficiency:** O(n√n), good for ranges like 1–500 or 1–10000. Prompt:

Code and output:

## ⬛ Optimized Prime Number Program (Square Root Method)

****

Ih \_ I **Explanation of Optimizations**

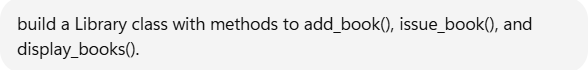
1. Why check up to √num?
   * If num is composite, it must have at least one factor ≤ √num.
   * Example: 36 = 6 × 6 → factors repeat after √36.
   * So, checking only up to √num avoids redundant checks and speeds up the process.
2. Efficiency improvement:
   * Naive prime check → check all numbers from 2 to num-1 → O(n²) for range n.
   * Optimized prime check (√num method) → check only up to square root → O(n√n).
   * Much faster, especially for larger ranges.
3. Example:
   * Checking if 997 is prime:
     + Naive → 995 checks.
     + Optimized → only 31 checks (since √997 ≈ 31).

⬛ Correctness: Same results as the naive method.

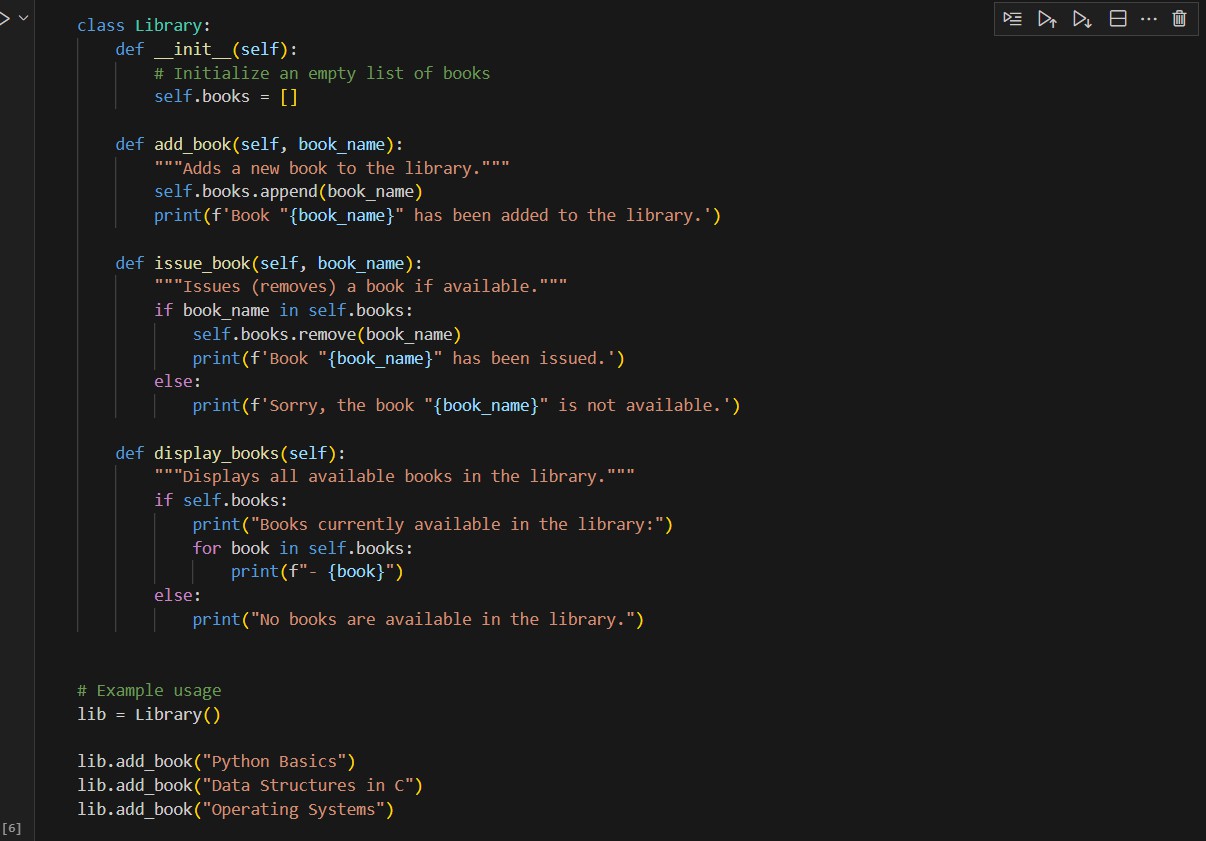
) Efficiency: Greatly reduced checks, faster execution for large ranges**.**

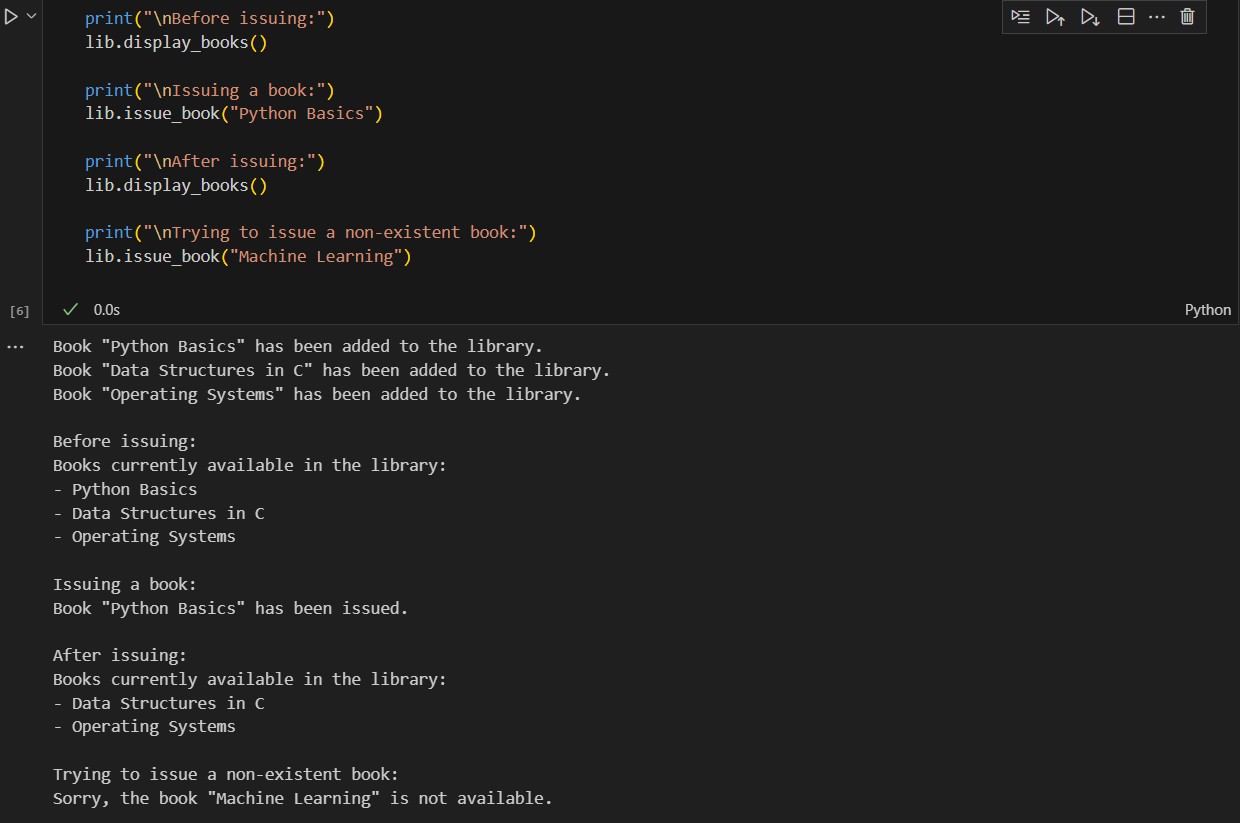
**Task-5:**

**Prompt:**

****

**Code with output:**

****



**Analyze if methods handle edge cases (e.g., issuing unavailable books):**

⬛ add\_book(book\_name)

* Correctness:
  + Works fine for normal cases.
  + Accepts any string and adds it.
* Edge Cases:
  + Adding the same book multiple times → It will allow duplicates since it uses a list.
    - Example: Adding "Python Basics" twice will show it twice.
  + Adding an empty string "" → It will still add, though meaningless.

ı . Improvement: Add validation (e.g., check for duplicates, reject empty names).

⬛ issue\_book(book\_name)

* Correctness:
  + If book exists in the list, it gets removed.
  + If not, prints a friendly message "not available".
* Edge Cases:
  + Issuing a book when the library is empty → Still handled (goes to the "not available" message).
  + Trying to issue a book that was never added → Correctly shows "not available".
  + If duplicates exist → It only removes the first occurrence, not all copies.

ı . Improvement: If multiple copies exist, handle count instead of removing only one blindly.

⬛ display\_books()

* Correctness:
  + Displays all books currently available.
  + If none, shows "No books are available".
* Edge Cases:
  + Works fine even if library is empty (safe check).
  + If duplicates exist, all are shown, which may look odd.

ı . Improvement: Consider showing books with counts (e.g., "Python Basics (2 copies)").

’‘ z Conclusion

* Current methods handle unavailable book issuing safely ⬛.
* Weaknesses: duplicates allowed, no copy-tracking, and no input validation. Prompt:



Code with output:

