LAB-6 ASSIGNEMENT

NAME : AKULA VIVEK

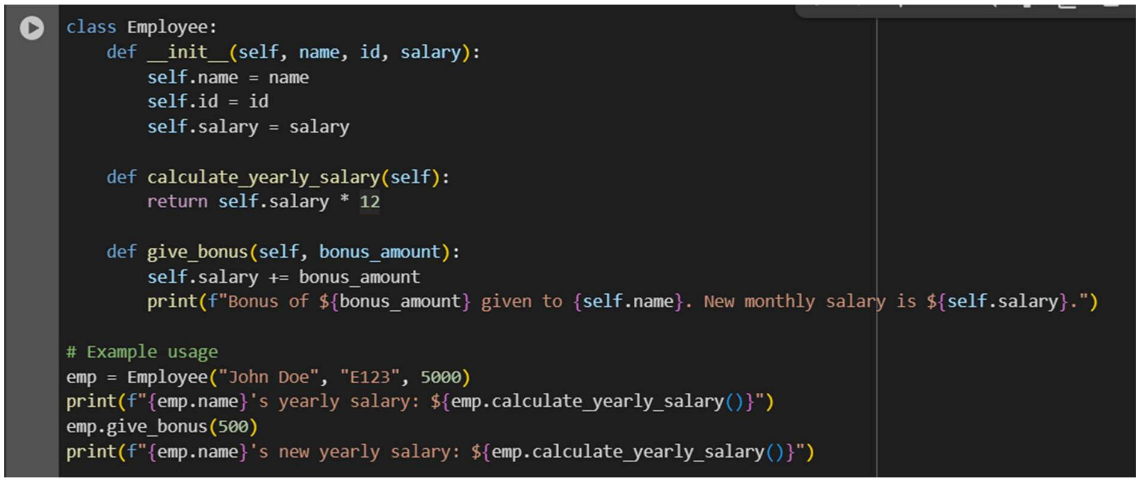
ROLL NO. : 2403A51258

BATCH : 11

Task Description #1 (Classes – Employee Management)

- Task: Use AI to create an Employee class with attributes (name, id, salary) and a method to calculate yearly salary.
- Instructions:
    - Prompt AI to generate the Employee class.
    - Analyze the generated code for correctness and structure.
    - Ask AI to add a method to give a bonus and recalculate salary.

Code:



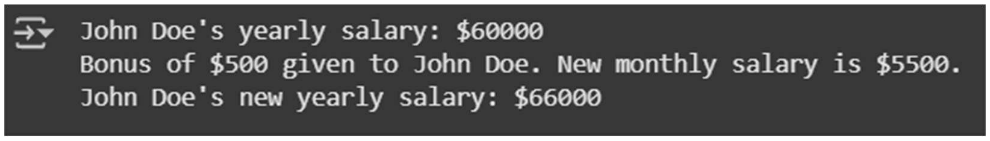
```python
class Employee:
    def __init__(self, name, id, salary):
        self.name = name
        self.id = id
        self.salary = salary

    def calculate_yearly_salary(self):
        return self.salary * 12

    def give_bonus(self, bonus_amount):
        self.salary += bonus_amount
        print(f"Bonus of ${bonus_amount} given to {self.name}. New monthly salary is ${self.salary}.")

# Example usage
emp = Employee("John Doe", "E123", 5000)
print(f"{emp.name}'s yearly salary: ${emp.calculate_yearly_salary()}")
emp.give_bonus(500)
print(f"{emp.name}'s new yearly salary: ${emp.calculate_yearly_salary()}")
```

Output:

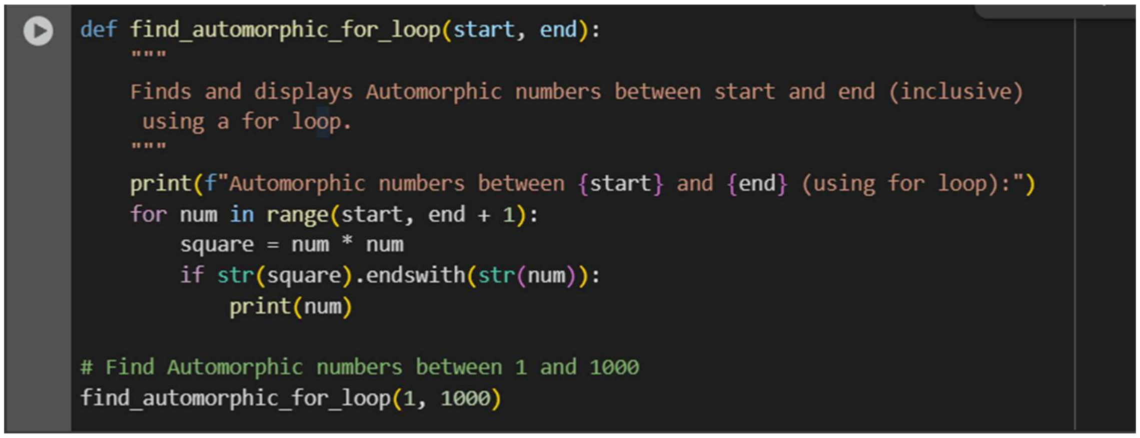


```
John Doe's yearly salary: $60000
Bonus of $500 given to John Doe. New monthly salary is $5500.
John Doe's new yearly salary: $66000
```

Task Description #2 (Loops – Automorphic Numbers in a Range)

- Task: Prompt AI to generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.
- Instructions:
    - Get AI-generated code to list Automorphic numbers using a for loop.
    - Analyze the correctness and efficiency of the generated logic.
    - Ask AI to regenerate using a while loop and compare both implementations.

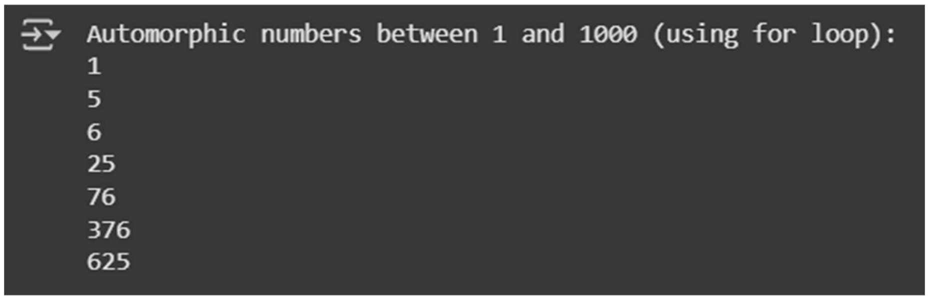
Code:



```python
def find_automorphic_for_loop(start, end):
    """
    Finds and displays Automorphic numbers between start and end (inclusive)
     using a for loop.
    """
    print(f"Automorphic numbers between {start} and {end} (using for loop):")
    for num in range(start, end + 1):
        square = num * num
        if str(square).endswith(str(num)):
            print(num)

# Find Automorphic numbers between 1 and 1000
find_automorphic_for_loop(1, 1000)
```

Output:



```
Automorphic numbers between 1 and 1000 (using for loop):
1
5
6
25
76
376
625
```

Explanation :

This code defines a Python function called find_automorphic_while_loop that finds and displays Automorphic numbers within a specified range using a while loop.

Here's a breakdown:

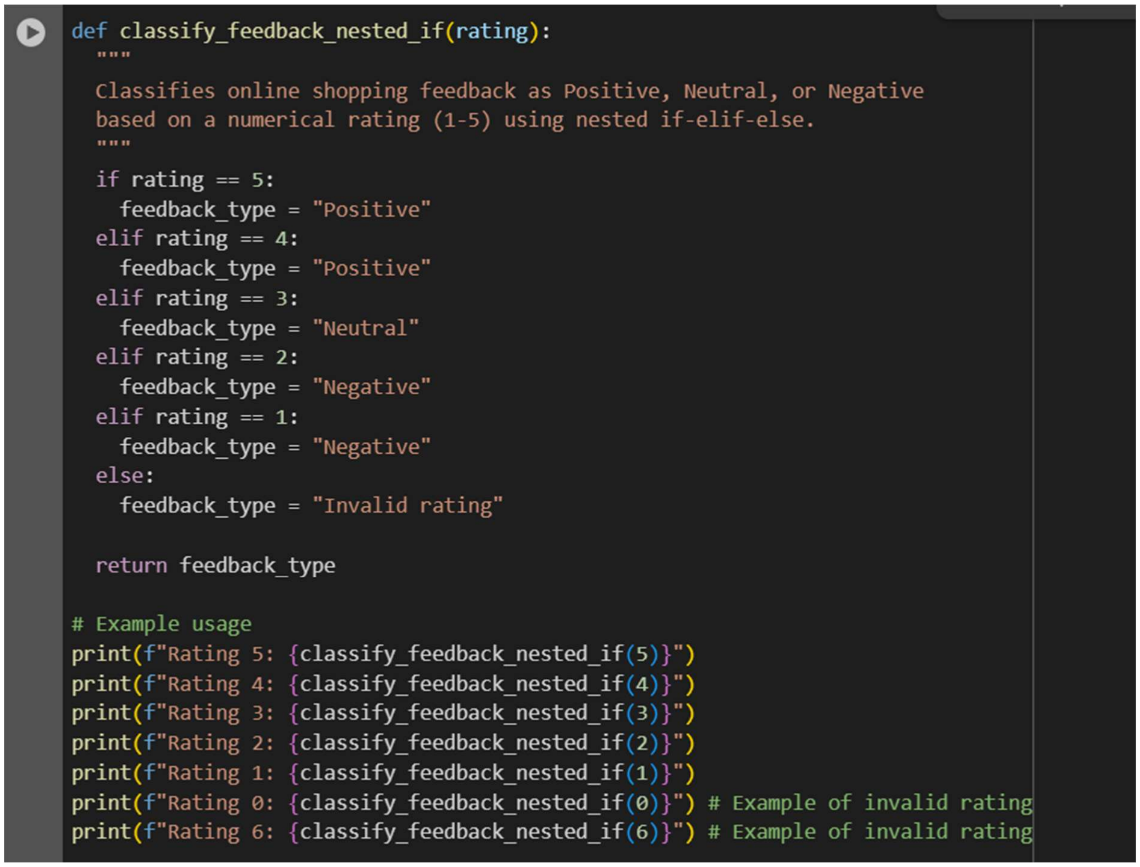
1. **def find_automorphic_while_loop(start, end):**: This defines the function find_automorphic_while_loop that takes two arguments: start and end, representing the beginning and end of the range to check for Automorphic numbers.
2. **print(f"Automorphic numbers between {start} and {end} (using while loop):")**: This line prints a header indicating the range being checked and that the while loop method is being used.
3. **num = start**: This line initializes a variable num with the start value of the range. This variable will be used as the current number being checked.
4. **while num <= end:**: This is the while loop condition. The code inside the loop will continue to execute as long as the value of num is less than or equal to the end value of the range.
5. **square = num * num**: Inside the loop, this line calculates the square of the current number num and stores it in the square variable.
6. **if str(square).endswith(str(num)):**: This is the core logic to check if a number is Automorphic.
    - str(square): Converts the square of the number to a string.
    - str(num): Converts the original number num to a string.
    - .endswith(str(num)): This string method checks if the string representation of the square ends with the string representation of the original num. If it does, the number is Automorphic.
7. **print(num)**: If the condition in the if statement is True (meaning the number is Automorphic), this line prints the Automorphic number.
8. **num += 1**: This line increments the value of num by 1. This is crucial for the while loop to eventually terminate; otherwise, it would run indefinitely.
9. **find_automorphic_while_loop(1, 1000)**: This line calls the function with the range 1 to 1000 to demonstrate its usage.

In essence, the code iterates through each number in the specified range using a while loop, calculates its square, and checks if the square ends with the original number to identify and print Automorphic numbers

Task Description #3 (Conditional Statements – Online Shopping Feedback Classification)

- Task: Ask AI to write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1–5).
- Instructions:
  - Generate initial code using nested if-elif-else.
  - Analyze correctness and readability.
  - Ask AI to rewrite using dictionary-based or match-case structure.

Code:

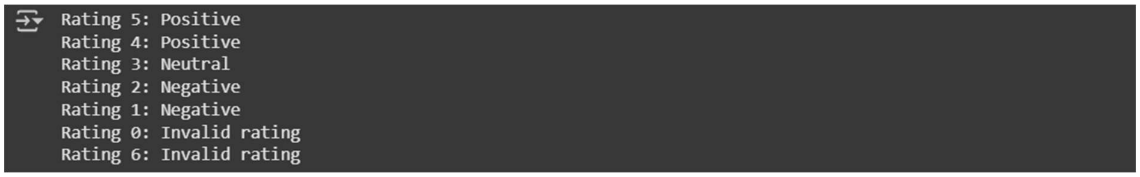


```
def classify_feedback_nested_if(rating):
  """
  Classifies online shopping feedback as Positive, Neutral, or Negative
  based on a numerical rating (1-5) using nested if-elif-else.
  """
  if rating == 5:
    feedback_type = "Positive"
  elif rating == 4:
    feedback_type = "Positive"
  elif rating == 3:
    feedback_type = "Neutral"
  elif rating == 2:
    feedback_type = "Negative"
  elif rating == 1:
    feedback_type = "Negative"
  else:
    feedback_type = "Invalid rating"

  return feedback_type

# Example usage
print(f"Rating 5: {classify_feedback_nested_if(5)}")
print(f"Rating 4: {classify_feedback_nested_if(4)}")
print(f"Rating 3: {classify_feedback_nested_if(3)}")
print(f"Rating 2: {classify_feedback_nested_if(2)}")
print(f"Rating 1: {classify_feedback_nested_if(1)}")
print(f"Rating 0: {classify_feedback_nested_if(0)}") # Example of invalid rating
print(f"Rating 6: {classify_feedback_nested_if(6)}") # Example of invalid rating
```

Output:



```
Rating 5: Positive
Rating 4: Positive
Rating 3: Neutral
Rating 2: Negative
Rating 1: Negative
Rating 0: Invalid rating
Rating 6: Invalid rating
```

Explanation :

classify_feedback_dictionary which classifies feedback using a dictionary lookup, and you can see the code and its output above in cell f2871fbb.

Comparing this dictionary-based approach to the previous if-elif-else version in cell 40d37b8a:

Readability: For this specific mapping of discrete values to categories, the dictionary approach is generally considered more readable and maintainable, as the mapping is clearly defined in one place.

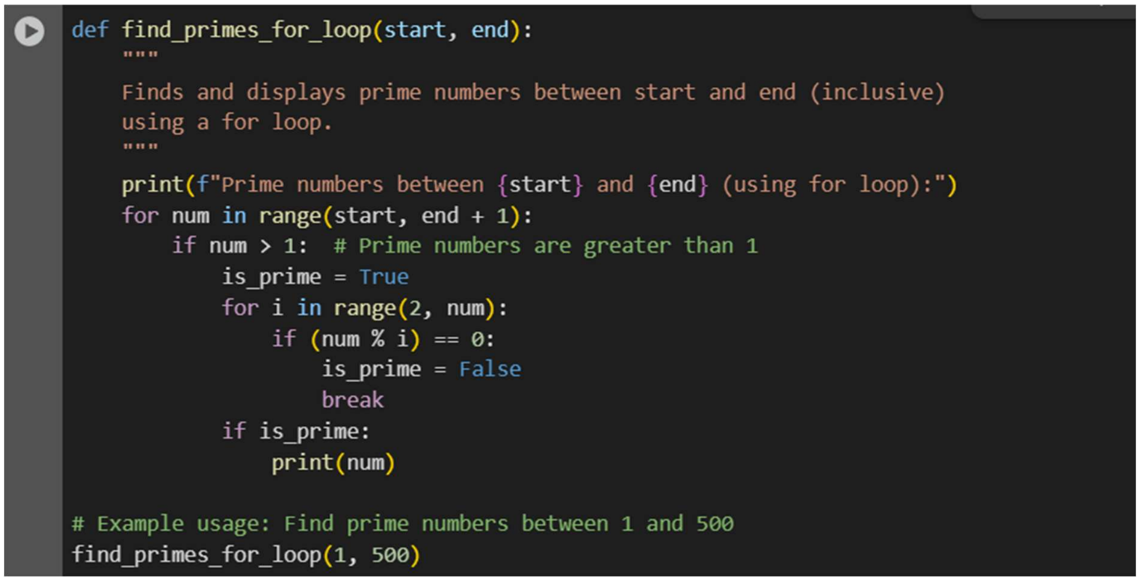
Conciseness: The dictionary approach is more concise.

Efficiency: For a small number of conditions, the performance difference is negligible. For a very large number of exact matches, dictionary lookups can be faster than a long chain of elif checks.

Task Description #4 (Loops – Prime Numbers in a Range)

- Task: Generate a function using AI that displays all prime numbers within a user-specified range (e.g., 1 to 500).
- Instructions:
    - Get AI-generated code to list all primes using a for loop.
    - Analyze the correctness and efficiency of the prime-checking logic.
    - Ask AI to regenerate an optimized version (e.g., using the square root method).

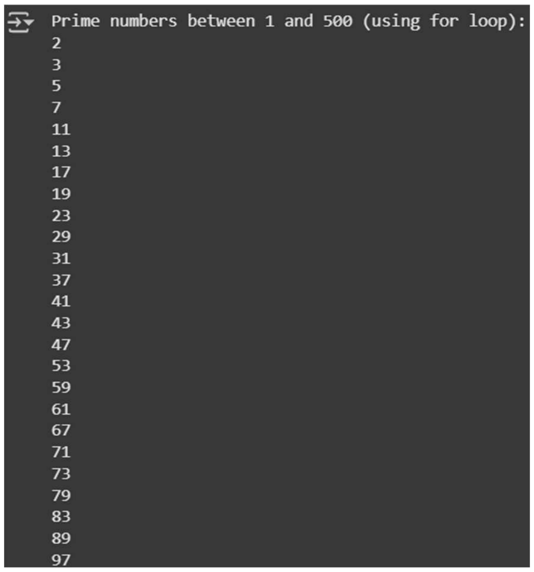
Code:



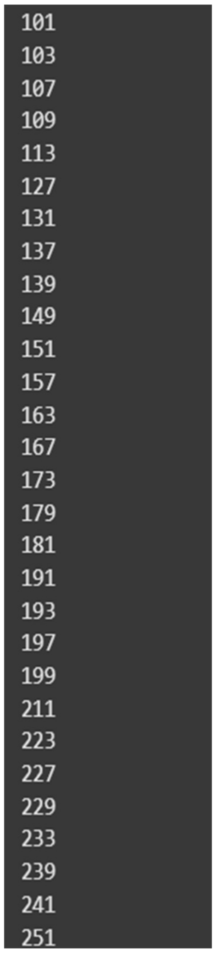
```
def find_primes_for_loop(start, end):
    """
    Finds and displays prime numbers between start and end (inclusive)
    using a for loop.
    """
    print(f"Prime numbers between {start} and {end} (using for loop):")
    for num in range(start, end + 1):
        if num > 1:  # Prime numbers are greater than 1
            is_prime = True
            for i in range(2, num):
                if (num % i) == 0:
                    is_prime = False
                    break
            if is_prime:
                print(num)

# Example usage: Find prime numbers between 1 and 500
find_primes_for_loop(1, 500)
```

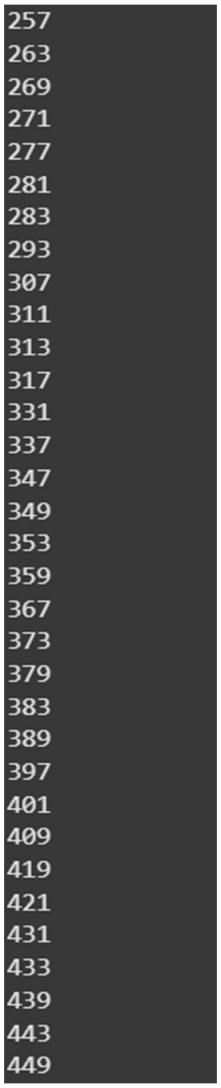
Output:



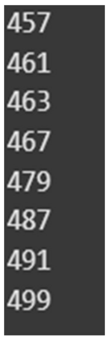
```
Prime numbers between 1 and 500 (using for loop):
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```



```
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
```

Explanation :

This code defines a function called find_primes_optimized that finds and displays prime numbers within a specified range using an optimized approach.

Here's a breakdown:

import math: This line imports the math module, which is needed to use the math.sqrt() function for calculating the square root.

def find_primes_optimized(start, end):: This defines the function find_primes_optimized that takes two arguments: start and end, representing the beginning and end of the range to check for prime numbers.

print(f"Prime numbers between {start} and {end} (optimized):"): This line prints a header indicating the range being checked and that the optimized method is being used.

for num in range(start, end + 1):: This is the outer loop that iterates through each number from start to end (inclusive).

if num > 1:: This condition checks if the current number num is greater than 1, because prime numbers are defined as numbers greater than 1.

is_prime = True: Inside the if num > 1: block, a boolean variable is_prime is initialized to True. This variable will be used to track whether the current number is prime. It is assumed to be prime until proven otherwise.

for i in range(2, int(math.sqrt(num)) + 1):: This is the inner loop, which is the core of the optimized primality test. It iterates through potential divisors i starting from 2 up to the integer part of the square root of num (inclusive). We only need to check up to the square root because if a number has a divisor larger than its square root, it must also have a divisor smaller than its square root.

if (num % i) == 0:: Inside the inner loop, this condition checks if num is perfectly divisible by i. If it is, it means num has a divisor other than 1 and itself, so it is not a prime number.

is_prime = False: If num is divisible by i, is_prime is set to False.

break: If a divisor is found, there's no need to check further for this num, so the break statement exits the inner loop.

if is_prime:: After the inner loop finishes (either by checking all potential divisors up to the square root or by finding a divisor and breaking), this condition checks if is_prime is still True.

print(num): If is_prime is True, it means the number num is prime, and it is printed to the console.

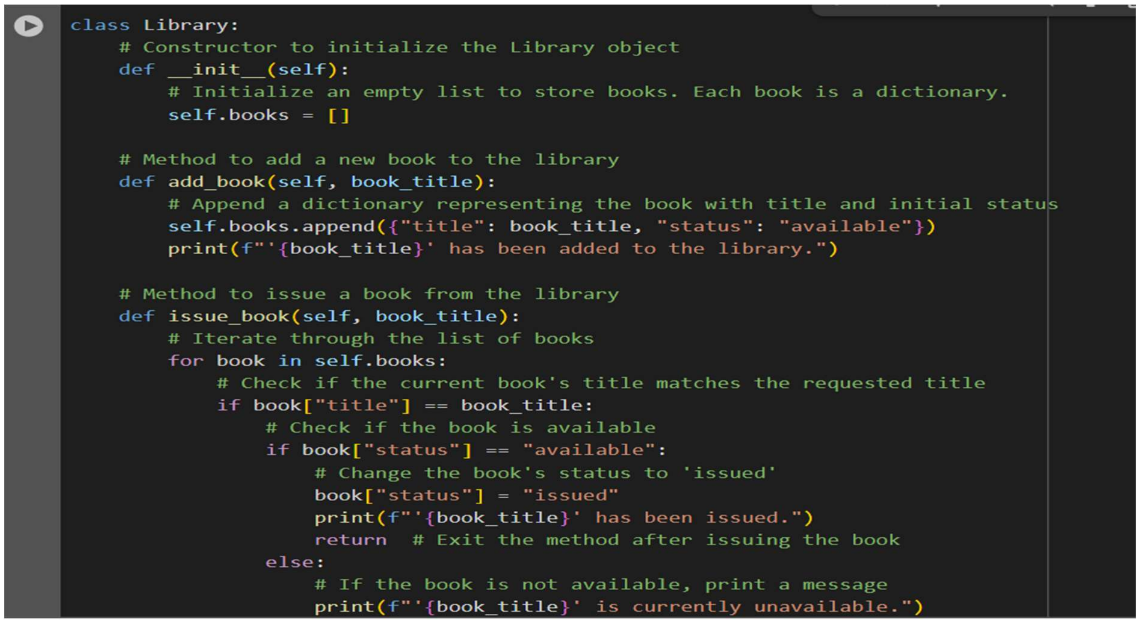
find_primes_optimized(1, 500): This line calls the function with the range 1 to 500 to demonstrate its usage.

In summary, this function efficiently identifies prime numbers within a given range by only checking for divisibility up to the square root of each number.

Task Description #5 (Classes – Library System)

- Task: Use AI to build a Library class with methods to add_book(), issue_book(), and display_books().
- Instructions:
  - Generate Library class code using AI.
  - Analyze if methods handle edge cases (e.g., issuing unavailable books).
  - Ask AI to add comments and documentation.

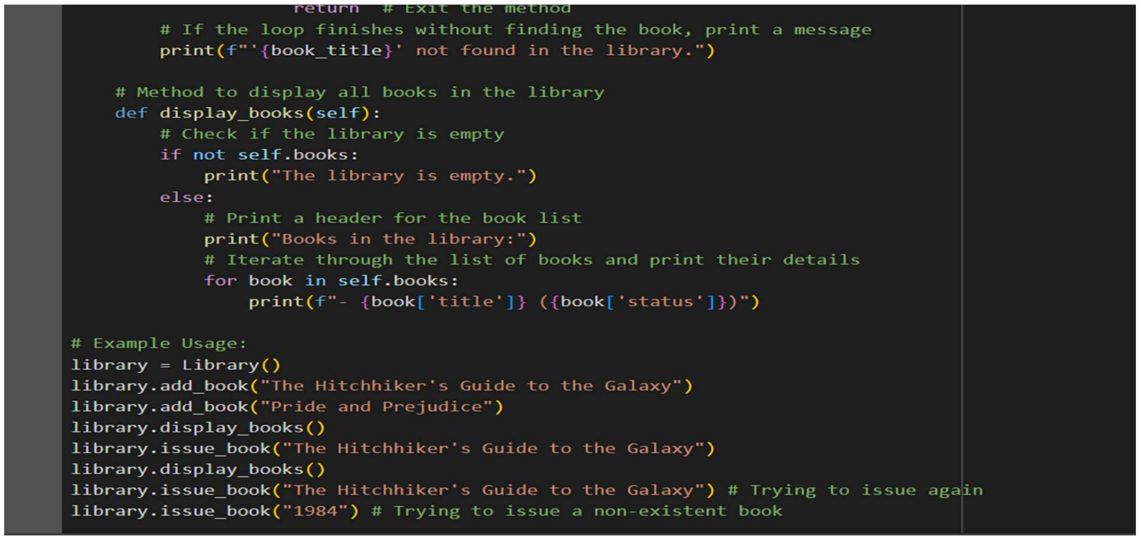
Code :



```
class Library:
    # Constructor to initialize the Library object
    def __init__(self):
        # Initialize an empty list to store books. Each book is a dictionary.
        self.books = []

    # Method to add a new book to the library
    def add_book(self, book_title):
        # Append a dictionary representing the book with title and initial status
        self.books.append({"title": book_title, "status": "available"})
        print(f"'{book_title}' has been added to the library.")

    # Method to issue a book from the library
    def issue_book(self, book_title):
        # Iterate through the list of books
        for book in self.books:
            # Check if the current book's title matches the requested title
            if book["title"] == book_title:
                # Check if the book is available
                if book["status"] == "available":
                    # Change the book's status to 'issued'
                    book["status"] = "issued"
                    print(f"'{book_title}' has been issued.")
                    return  # Exit the method after issuing the book
                else:
                    # If the book is not available, print a message
                    print(f"'{book_title}' is currently unavailable.")
```

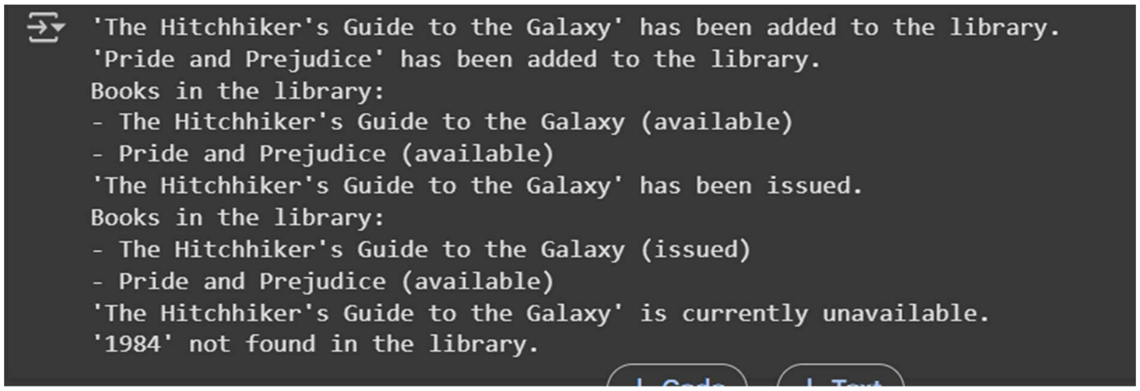


```
                    return  # Exit the method
        # If the loop finishes without finding the book, print a message
        print(f"'{book_title}' not found in the library.")

    # Method to display all books in the library
    def display_books(self):
        # Check if the library is empty
        if not self.books:
            print("The library is empty.")
        else:
            # Print a header for the book list
            print("Books in the library:")
            # Iterate through the list of books and print their details
            for book in self.books:
                print(f"- {book['title']} ({book['status']})")

# Example Usage:
library = Library()
library.add_book("The Hitchhiker's Guide to the Galaxy")
library.add_book("Pride and Prejudice")
library.display_books()
library.issue_book("The Hitchhiker's Guide to the Galaxy")
library.display_books()
library.issue_book("The Hitchhiker's Guide to the Galaxy") # Trying to issue again
library.issue_book("1984") # Trying to issue a non-existent book
```

Output :



```
'The Hitchhiker's Guide to the Galaxy' has been added to the library.
'Pride and Prejudice' has been added to the library.
Books in the library:
- The Hitchhiker's Guide to the Galaxy (available)
- Pride and Prejudice (available)
'The Hitchhiker's Guide to the Galaxy' has been issued.
Books in the library:
- The Hitchhiker's Guide to the Galaxy (issued)
- Pride and Prejudice (available)
'The Hitchhiker's Guide to the Galaxy' is currently unavailable.
'1984' not found in the library.
```

Exaplanation :

This part of the code demonstrates how to create and interact with a Library object using the methods defined in the Library class:

1. **library = Library()**: This line creates an instance of the Library class and assigns it to the variable library. This initializes an empty library.
2. **library.add_book("The Hitchhiker's Guide to the Galaxy")**: This line calls the add_book method on the library object to add the book titled "The Hitchhiker's Guide to the Galaxy" to the library.
3. **library.add_book("Pride and Prejudice")**: This line calls the add_book method again to add another book titled "Pride and Prejudice" to the library.
4. **library.display_books()**: This line calls the display_books method to show the current list of books in the library and their statuses. At this point, both books should be listed as "available".

5. **library.issue_book("The Hitchhiker's Guide to the Galaxy")**: This line calls the issue_book method to issue "The Hitchhiker's Guide to the Galaxy". The method will find the book and change its status to "issued".
6. **library.display_books()**: This line calls display_books again to show the updated list of books. "The Hitchhiker's Guide to the Galaxy" should now be listed as "issued", while "Pride and Prejudice" remains "available".
7. **library.issue_book("The Hitchhiker's Guide to the Galaxy")**: This line attempts to issue "The Hitchhiker's Guide to the Galaxy" again. Since its status is already "issued", the issue_book method will detect this and print the "currently unavailable" message.
8. **library.issue_book("1984")**: This line attempts to issue a book titled "1984". Since this book was never added to the library, the issue_book method will not find it and will print the "not found in the library" message.

This example usage demonstrates the basic workflow of adding books, issuing books, and viewing the library's contents, as well as how the issue_book method handles cases where the book is unavailable or not found.

--END--