HALLTICKET NO:-2403A51317

BATCH NO:- 13

ASSIGNMENT:-6.1

DATE :-08.09.2025

TASK 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:

o Prompt AI to generate the Employee class.

o Analyze the generated code for correctness and structure.

o Ask AI to add a method to give a bonus and recalculate salary.

Expected Output #1:

• A class with constructor, display_details(), and calculate_bonus() methods

CODE:-

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

    def display_details(self):
        print(f"Name: {self.name}, ID: {self.id}, Salary: {self.salary}")

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

    def give_bonus(self, bonus_percentage):
        bonus_amount = self.salary * (bonus_percentage / 100)
        self.salary += bonus_amount
        print(f"Bonus of {bonus_amount} applied. New salary: {self.salary}")
```

OUTPUT:-

```
Name: Alice, ID: E123, Salary: 5000
Yearly Salary: 60000
Bonus of 500.0 applied. New salary: 5500.0
Name: Alice, ID: E123, Salary: 5500.0
```

Code Analysis

The provided Python code effectively addresses the task of creating an **Employee** class. Let's break down its structure and correctness.

Class Structure:-

- _init__(self, name, id, salary): This is the constructor method. It's automatically called when a new Employee object is created. It initializes three key attributes for each employee: name, id, and salary.
- display_details(self): This method prints the employee's details in a clear, formatted way.
- yearly_salary(self): This method calculates and returns the employee's total salary for a year by multiplying the monthly salary by 12.
- calculate_bonus(self, bonus_percentage): This method handles the bonus calculation and salary update. It takes a bonus_percentage as an argument, calculates the bonus amount, adds it to the existing salary attribute, and then displays the bonus amount and the new salary. This demonstrates how a method can both perform a calculation and modify an object's state (its attributes).

TASK 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:

o Get AI-generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

o Ask AI to regenerate using a while loop and compare both implementations.

Expected Output #2:

• Correct implementation that lists Automorphic numbers using both loop types, with explanation.

CODE:-

```
def find_automorphic_while(start, end):
    """
    Finds automorphic numbers within a given range using a while loop.

    An automorphic number is a number whose square ends in the same digits as the number itself.
    For example, 25 is an automorphic number because 25*25 = 625, and 625 ends in 25.
    """
    print(f"\nAutomorphic numbers between {start} and {end} (using while loop):")
    num = start
    while num <= end:
        square = num * num
        if str(square).endswith(str(num)):
            print(num)
        num += 1

# Find automorphic numbers between 1 and 1000
find_automorphic_while(1, 1000)
```

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

CODE ANALYSIS

Here is a Python function that finds Automorphic numbers between 1 and 1000 using a for loop:

Great! We have successfully generated the code to find automorphic numbers using a for loop.

Now, let's create a function that does the same using a while loop and then we can compare the two implementations as you requested.

We have now generated and executed both the for loop and whileloop implementations for finding automorphic numbers.

Both functions correctly identify the automorphic numbers within the specified range. The core logic for checking if a number is automorphic (str(square).endswith(str(num))) is the same in both.

The main difference lies in how the iteration is handled:

- **for loop:** This is generally considered more concise and Pythonic for iterating over a known sequence or range. The loop handles the initialization and incrementing of the counter variable automatically (for num in range(start, end + 1):).
- **while loop:** This provides more flexibility when the number of iterations is not known in advance or when the loop condition is more complex. In this specific case, we need to manually initialize the counter (num = start) and increment it within the loop (num += 1).

For this particular task, the for loop implementation is slightly more straightforward due to iterating over a fixed range. However, both are valid and produce the same result.

TASK 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:

o Generate initial code using nested if-elif-else.

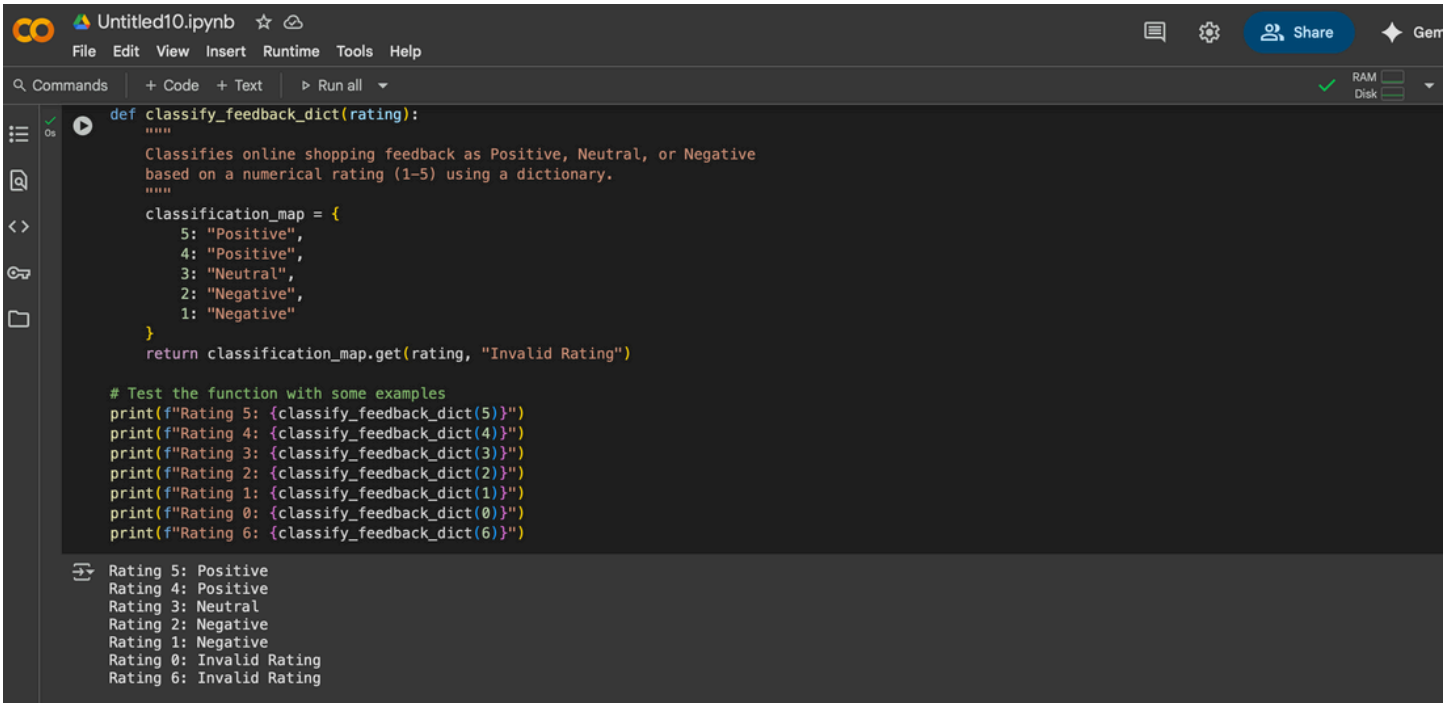
o Analyze correctness and readability.

o Ask AI to rewrite using dictionary-based or match-case structure.

Expected Output #3:

• Feedback classification function with explanation and an alternative approach

CODE:-



CODE ANALYSIS

.. Here is a Python function that classifies online shopping feedback using nested if-elif-else statements based on a numerical rating from 1 to 5:

Great! We have successfully created and tested the feedback classification function using nested if-elif-else statements.

Now, as you requested, let's explore an alternative approach using a dictionary-based structure (or match-case if you prefer, which is available in Python 3.10+). This can sometimes make the code more readable and maintainable, especially with more conditions.

Here's an example using a dictionary to map ratings to classifications:

- **Nested if-elif-else:** This is a straightforward and easy-to-understand approach for handling a relatively small number of conditional checks. However, for a larger number of conditions, it can become less readable and potentially harder to maintain.
- **Dictionary-based:** This approach uses a dictionary to map the possible input values (ratings) to their corresponding output values (classifications). This can be more concise and readable, especially when you have many distinct input values and their associated outputs. It also makes it easier to add or modify classifications by simply updating the dictionary. The .get() method is used to handle cases where the rating is not in the dictionary, providing a default "Invalid Rating" message

TASK 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:

o Get AI-generated code to list all primes using a for loop.

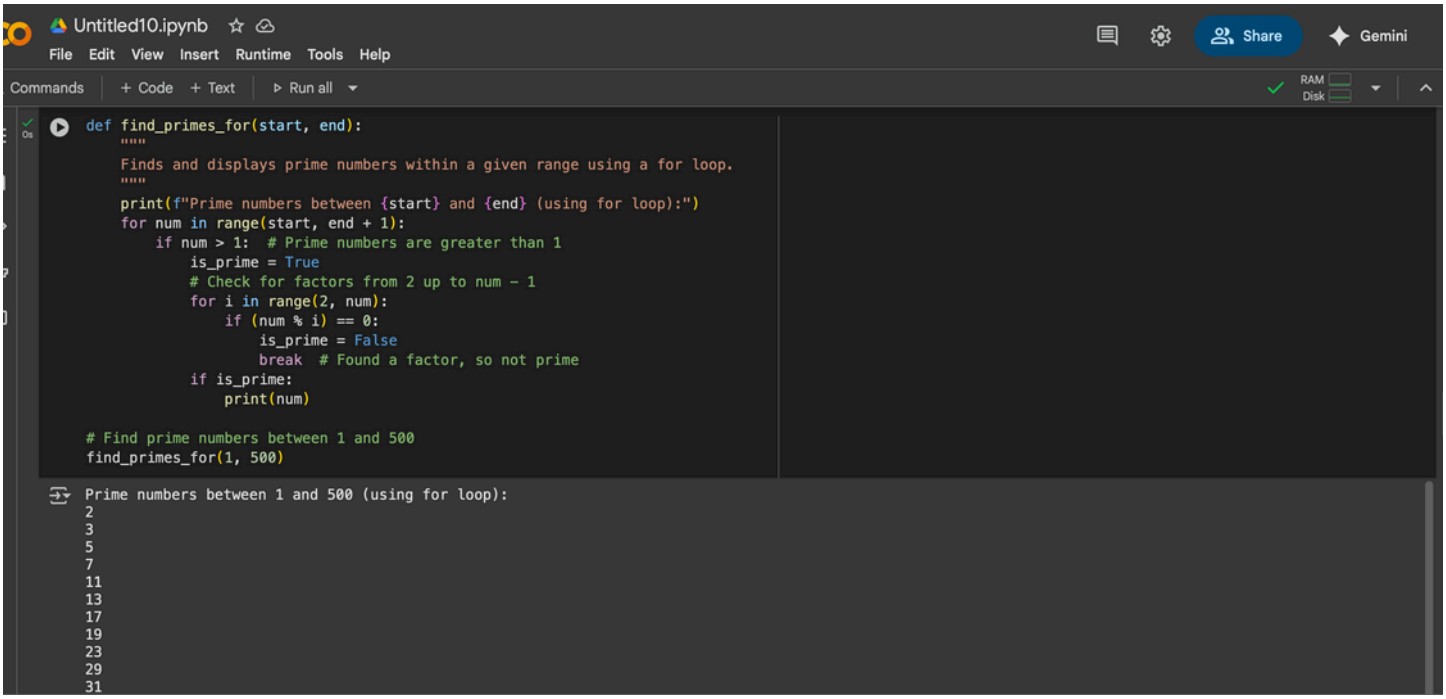
o Analyze the correctness and efficiency of the prime-checking logic.

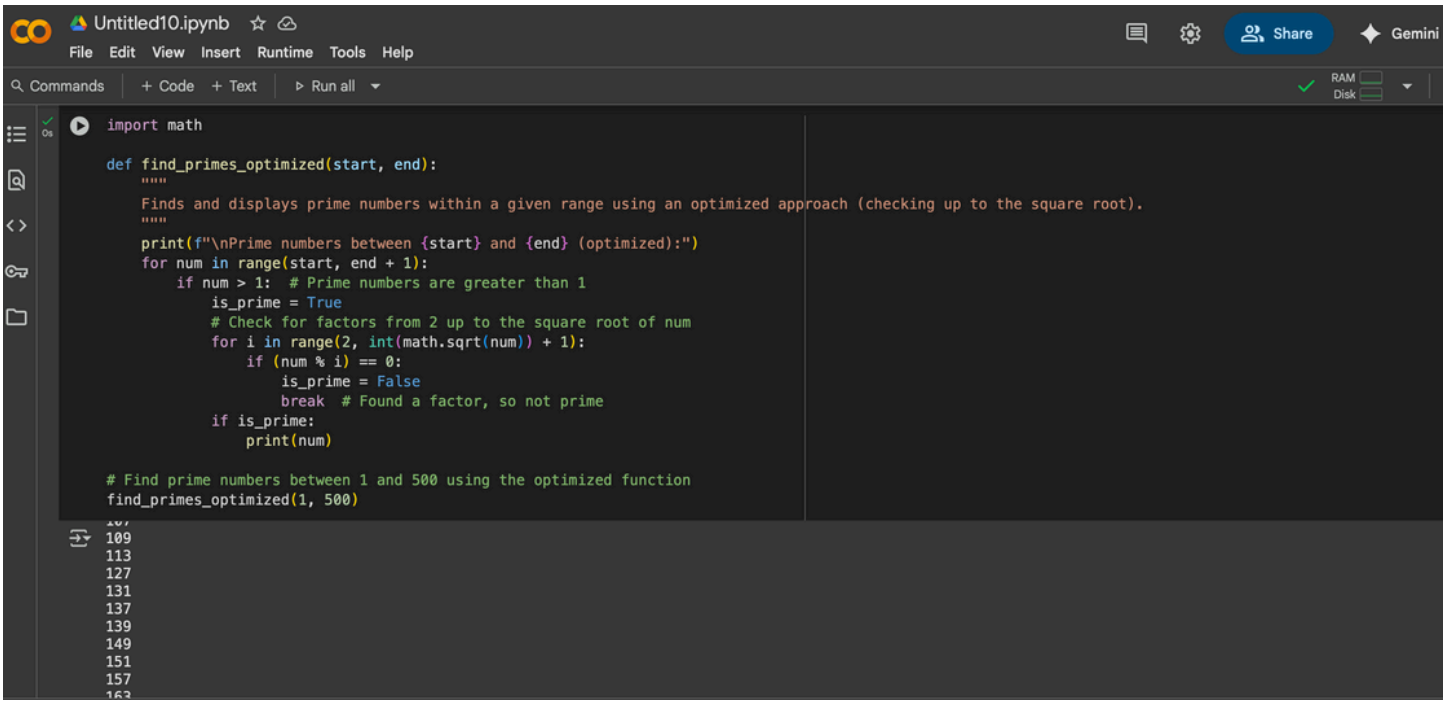
o Ask AI to regenerate an optimized version (e.g., using the square root method).

Expected Output #4:

• Python program that lists all prime numbers within a given range, with an optimized version and explanation.

CODE:-





CODE ANALYSIS:-

Here is a Python function that finds and displays prime numbers within a user-specified range (e.g., 1 to 500) using a for loop:

Great! We have successfully generated and executed the code to find prime numbers within the range of 1 to 500 using a for loop.

Let's analyze the correctness and efficiency of the current prime-checking logic.

Correctness: The logic is correct. It correctly identifies prime numbers by checking for divisibility by all integers from 2 up to num - 1. If no divisors are found, the number is prime.

Efficiency: The current approach checks for divisibility by every number from 2 up to num - 1. This can be inefficient for larger numbers. For example, to check if 100 is prime, it would check divisibility by 2, 3, 4, ..., 99. However, we only need to check for divisors up to the square root of the number. If a number n has a divisor i greater than its square root, it must also have a divisor j less than its square root (since i * j = n).

Now, as you requested, let's regenerate an optimized version of this function that uses the square root method to improve efficiency.

We have now generated and executed both the basic and the optimized functions for finding prime numbers within the range of 1 to 500.

Here's a comparison of the two approaches:

- **Basic Version (find_primes_for):** This version checks for divisibility by all numbers from 2 up to num - 1. While correct, it is less efficient, especially for larger numbers, as it performs unnecessary checks.
- **Optimized Version (find_primes_optimized):** This version incorporates the optimization of checking for divisibility only up to the square root of num. This significantly reduces the number of checks required, making the function much more efficient for larger ranges. We imported the math module to use the math.sqrt() function for calculating the square root.

TASK 5:-

(Classes – Library System)

• Task: Use AI to build a Library class with methods to add_book(), issue_book(), and display_books().

• Instructions:

o Generate Library class code using AI.

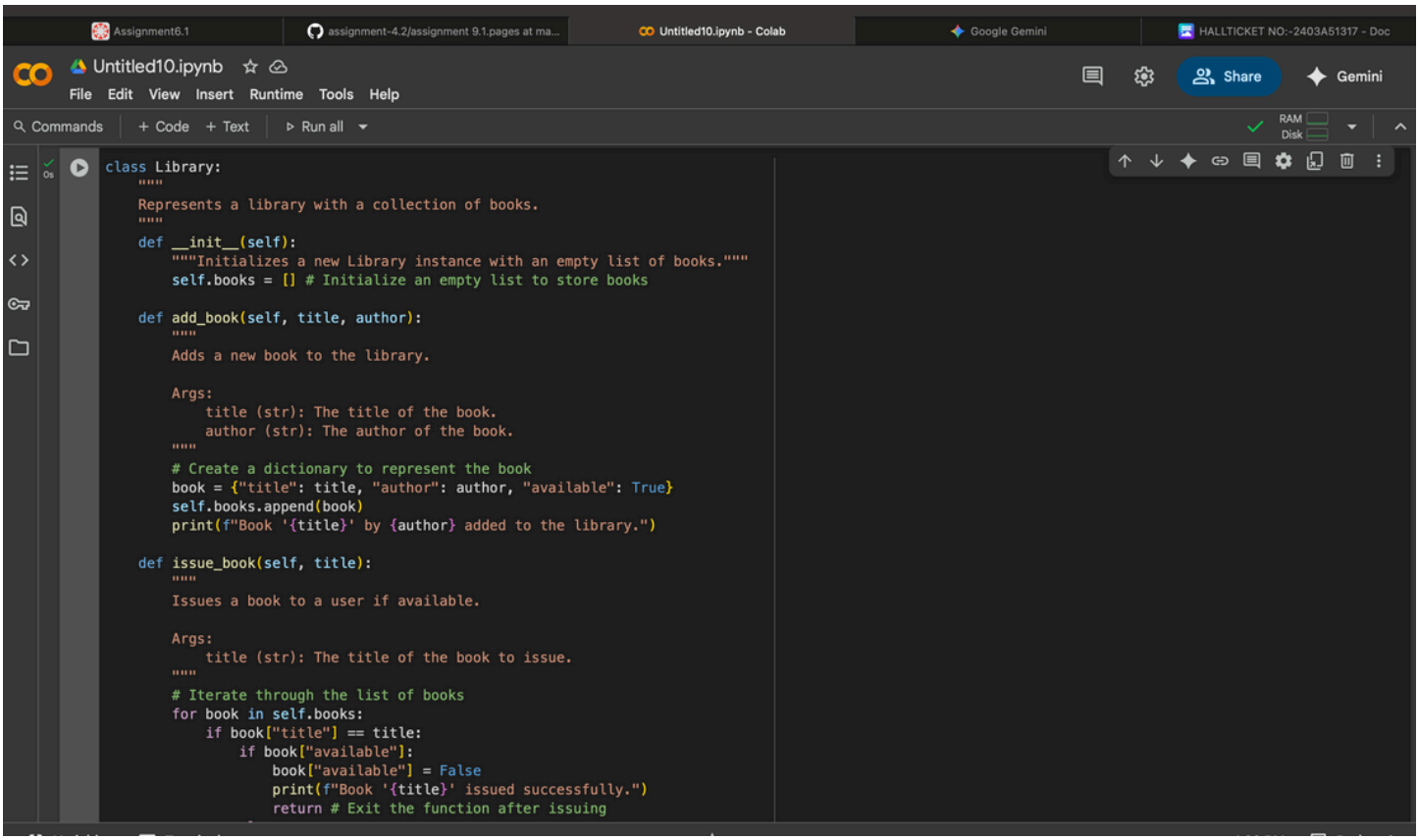
o Analyze if methods handle edge cases (e.g., issuing unavailable books).

o Ask AI to add comments and documentation.

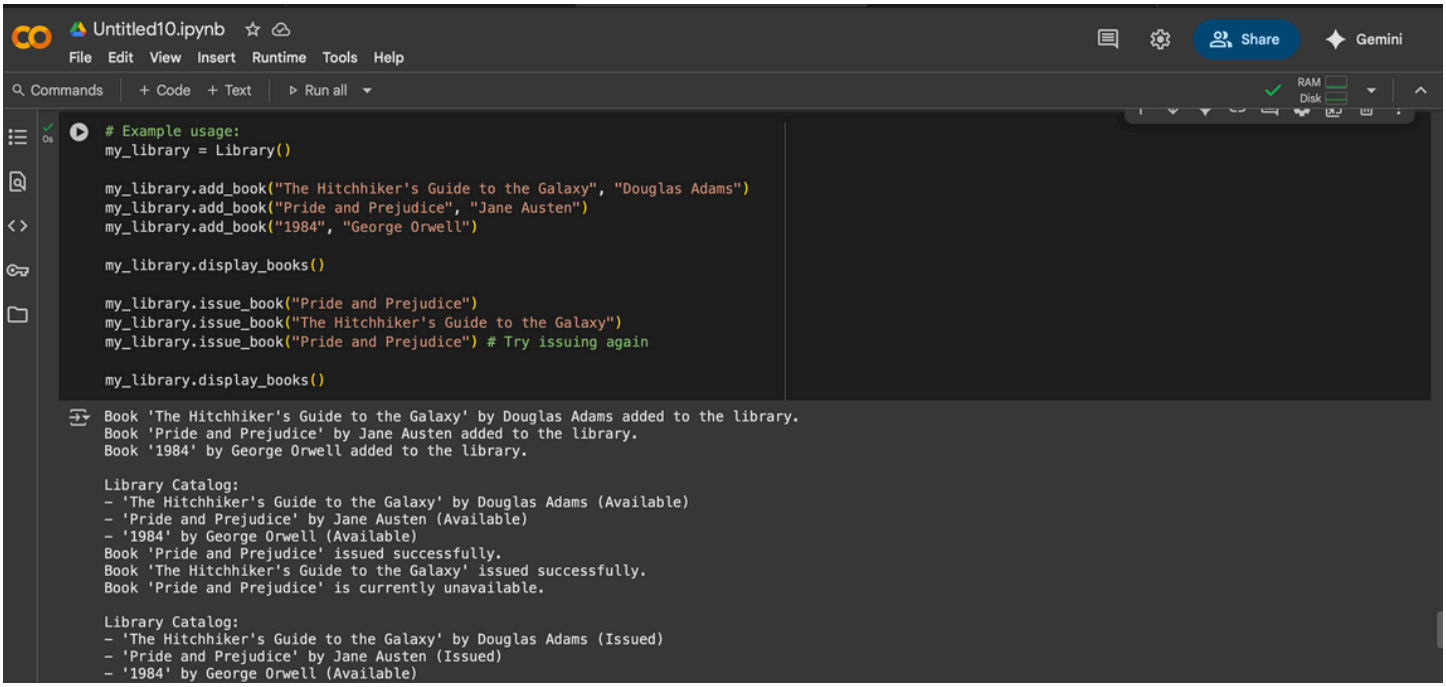
Expected Output #5:

• Library class with all methods, inline comments, and explanation.

CODE


Assignment6.1
assignment-4.2/assignment 9.1.pages at ma...
Untitled10.ipynb - Colab
Google Gemini
HALLTICKET NO:-2403A51317 - Doc
Untitled10.ipynb
File Edit View Insert Runtime Tools Help
Share
Gemini
Commands
+ Code
+ Text
Run all
RAM
Disk
class Library:
"""
Represents a library with a collection of books.
"""
def __init__(self):
"""Initializes a new Library instance with an empty list of books."""
self.books = [] # Initialize an empty list to store books
def add_book(self, title, author):
"""
Adds a new book to the library.
Args:
title (str): The title of the book.
author (str): The author of the book.
"""
# Create a dictionary to represent the book
book = {"title": title, "author": author, "available": True}
self.books.append(book)
print(f"Book '{title}' by {author} added to the library.")
def issue_book(self, title):
"""
Issues a book to a user if available.
Args:
title (str): The title of the book to issue.
"""
# Iterate through the list of books
for book in self.books:
if book["title"] == title:
if book["available"]:
book["available"] = False
print(f"Book '{title}' issued successfully.")
return # Exit the function after issuing


else:
print(f"Book '{title}' is currently unavailable.")
return # Exit the function if book is unavailable
# If the loop completes without finding the book
print(f"Book '{title}' not found in the library.")
def display_books(self):
"""
Displays all books in the library and their availability.
"""
if not self.books:
print("The library is empty.")
return # Exit the function if the library is empty
print("\nLibrary Catalog:")
# Iterate through the books and print their details
for book in self.books:
status = "Available" if book["available"] else "Issued"
print(f"- '{book['title']}' by {book['author']} ({status})")


Untitled10.ipynb
File Edit View Insert Runtime Tools Help
Share
Gemini
Commands
+ Code
+ Text
Run all
RAM
Disk
# Example usage:
my_library = Library()
my_library.add_book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams")
my_library.add_book("Pride and Prejudice", "Jane Austen")
my_library.add_book("1984", "George Orwell")
my_library.display_books()
my_library.issue_book("Pride and Prejudice")
my_library.issue_book("The Hitchhiker's Guide to the Galaxy")
my_library.issue_book("Pride and Prejudice") # Try issuing again
my_library.display_books()
Book 'The Hitchhiker's Guide to the Galaxy' by Douglas Adams added to the library.
Book 'Pride and Prejudice' by Jane Austen added to the library.
Book '1984' by George Orwell added to the library.
Library Catalog:
- 'The Hitchhiker's Guide to the Galaxy' by Douglas Adams (Available)
- 'Pride and Prejudice' by Jane Austen (Available)
- '1984' by George Orwell (Available)
Book 'Pride and Prejudice' issued successfully.
Book 'The Hitchhiker's Guide to the Galaxy' issued successfully.
Book 'Pride and Prejudice' is currently unavailable.
Library Catalog:
- 'The Hitchhiker's Guide to the Galaxy' by Douglas Adams (Issued)
- 'Pride and Prejudice' by Jane Austen (Issued)
- '1984' by George Orwell (Available)

CODE ANALYSIS

Here is the Python code for a Libraryclass with the add_book(), issue_book(), and display_books()methods:

Great! We have successfully created the Library class with add_book, issue_book, and display_books methods, and we've run some example code to see it in action.

Let's analyze how the methods handle some edge cases, as you requested:

- *add_book(): This method currently adds any book regardless of whether it already exists. An edge case here could be adding a check to prevent duplicate books based on title and author.*
- *issue_book(): The code correctly handles the edge case of trying to issue a book that is not found in the library by printing "Book '{title}' not found in the library.". It also handles the case of trying to issue a book that is already issued by printing "Book '{title}' is currently unavailable.".*
- *display_books(): This method correctly handles the edge case of an empty library by checking if not self.books: and printing "The library is empty.".*