AI-Ass-6.3

Name:T.shivani

Ht.No:2303A51312

Batch:05

Task Description #1: Classes (Student Class)

Scenario

You are developing a simple student information management module.

Task

• Use an AI tool (GitHub Copilot / Cursor AI / Gemini) to complete a Student class.

• The class should include attributes such as name, roll number, and branch.

• Add a method display_details() to print student information.

• Execute the code and verify the output.

• Analyze the code generated by the AI tool for correctness and clarity.

Expected Output #1

• A Python class with a constructor (__init__) and a display_details() method.

• Sample object creation and output displayed on the console.

• Brief analysis of AI-generated code.

Code:

```
class Student:
    def __init__(self, name, roll_number, branch):
        self.name = name
        self.roll_number = roll_number
        self.branch = branch

    def display_details(self):
        print(f"Name: {self.name}")
```

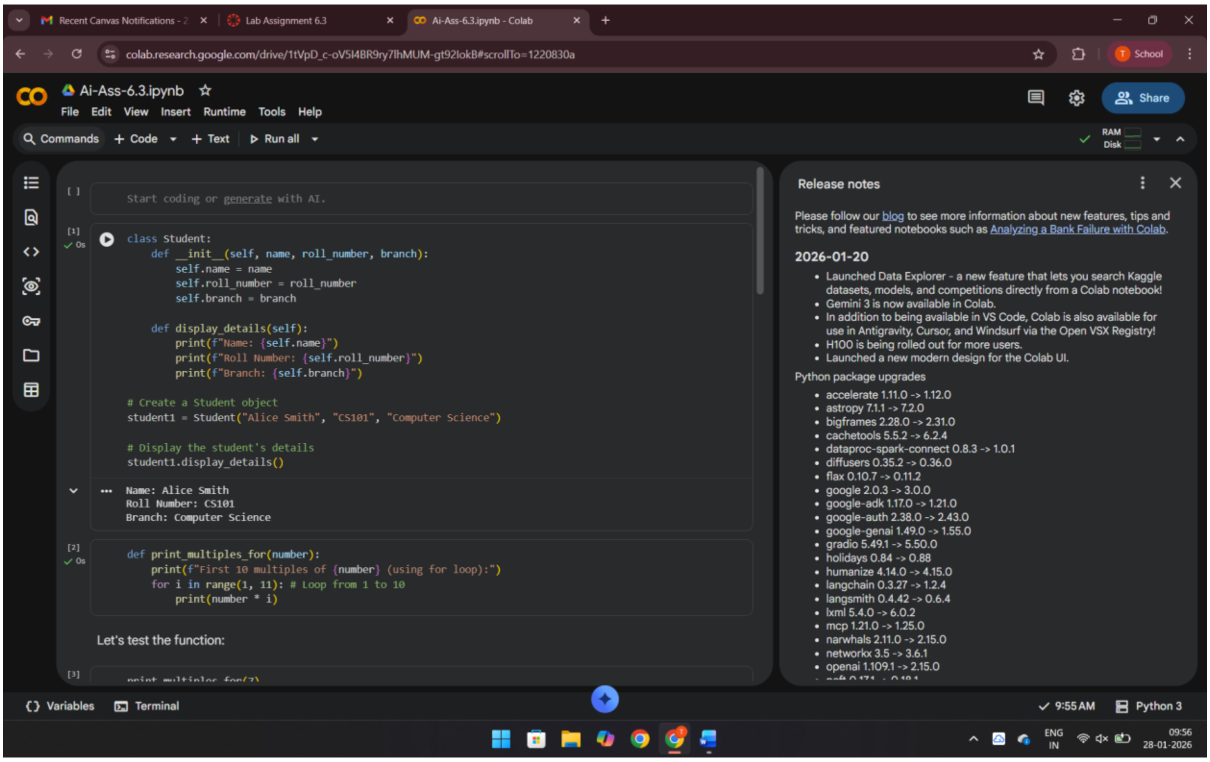
```
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")


# Create a Student object
student1 = Student("Alice Smith", "CS101", "Computer Science")


# Display the student's details
student1.display_details()
```

Output:

Name: Alice Smith

Roll Number: CS101

Branch: Computer Science



Explanation: the code defines a Student class with name, roll_number, and branch attributes. The display_details method prints these attributes. An instance of Student is created, and its details are displayed.

Task Description #2: Loops (Multiples of a Number)

Scenario

You are writing a utility function to display multiples of a given number.

Task

• Prompt the AI tool to generate a function that prints the first 10 multiples of a given number

using a loop.

• Analyze the generated loop logic.

• Ask the AI to generate the same functionality using another controlled looping structure (e.g.,

while instead of for).

Expected Output #2

• Correct loop-based Python implementation.

• Output showing the first 10 multiples of a number.

• Comparison and analysis of different looping approaches.

Code:
```
def print_multiples_for(number):
    print(f"First 10 multiples of {number} (using for loop):")
    for i in range(1, 11): # Loop from 1 to 10
        print(number * i)
```

Output: print_multiples_for(7)

First 10 multiples of 7 (using for loop):

7

14

21

28

35

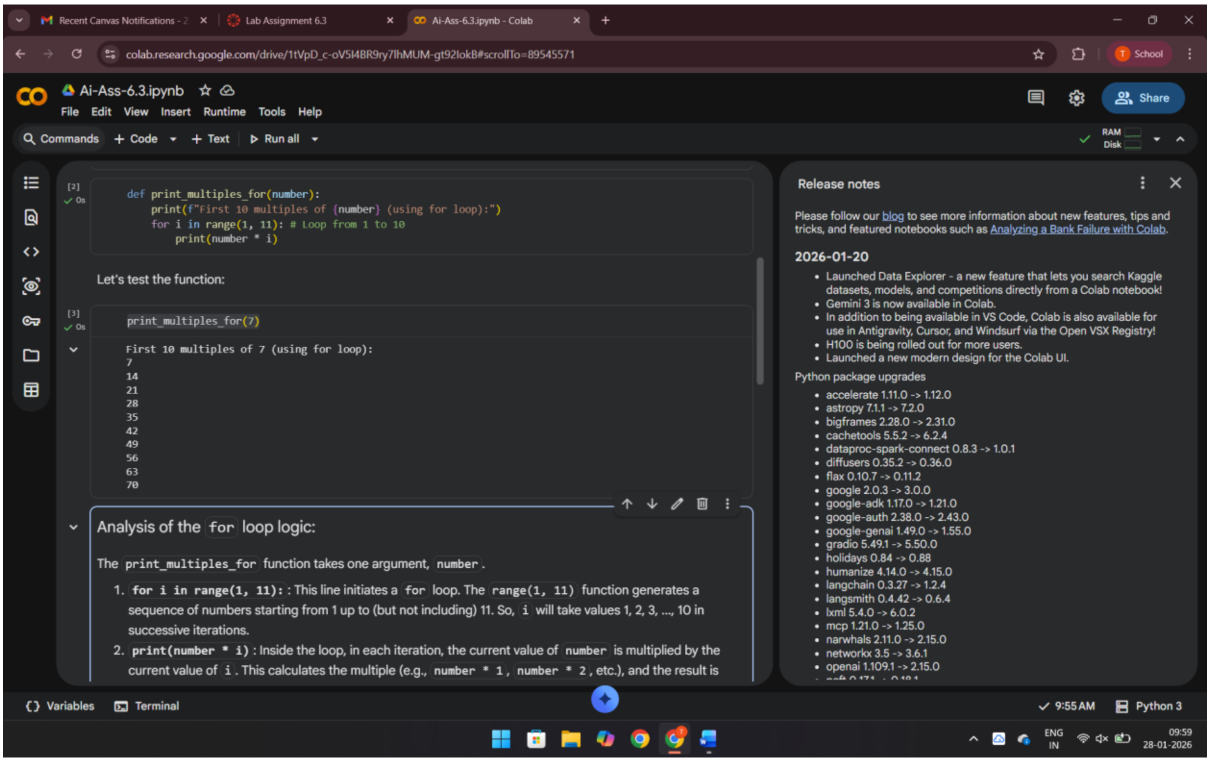
42

49

56

63

70



Explanation: Both functions, print_multiples_for and print_multiples_while, achieve the same goal: printing the first 10 multiples of a given number. The for loop version iterates a predefined number of times (10 times using range(1, 11)), while the while loop version uses a counter (count) and continues looping as long as the counter is less than or equal to 10.

Task Description #3: Conditional Statements (Age Classification)

Scenario

You are building a basic classification system based on age.

Task

• Ask the AI tool to generate nested if-elif-else conditional statements to classify age groups

(e.g., child, teenager, adult, senior).

• Analyze the generated conditions and logic.

• Ask the AI to generate the same classification using alternative conditional structures (e.g.,

simplified conditions or dictionary-based logic).

Expected Output #3

• A Python function that classifies age into appropriate groups.

• Clear and correct conditional logic.

• Explanation of how the conditions work

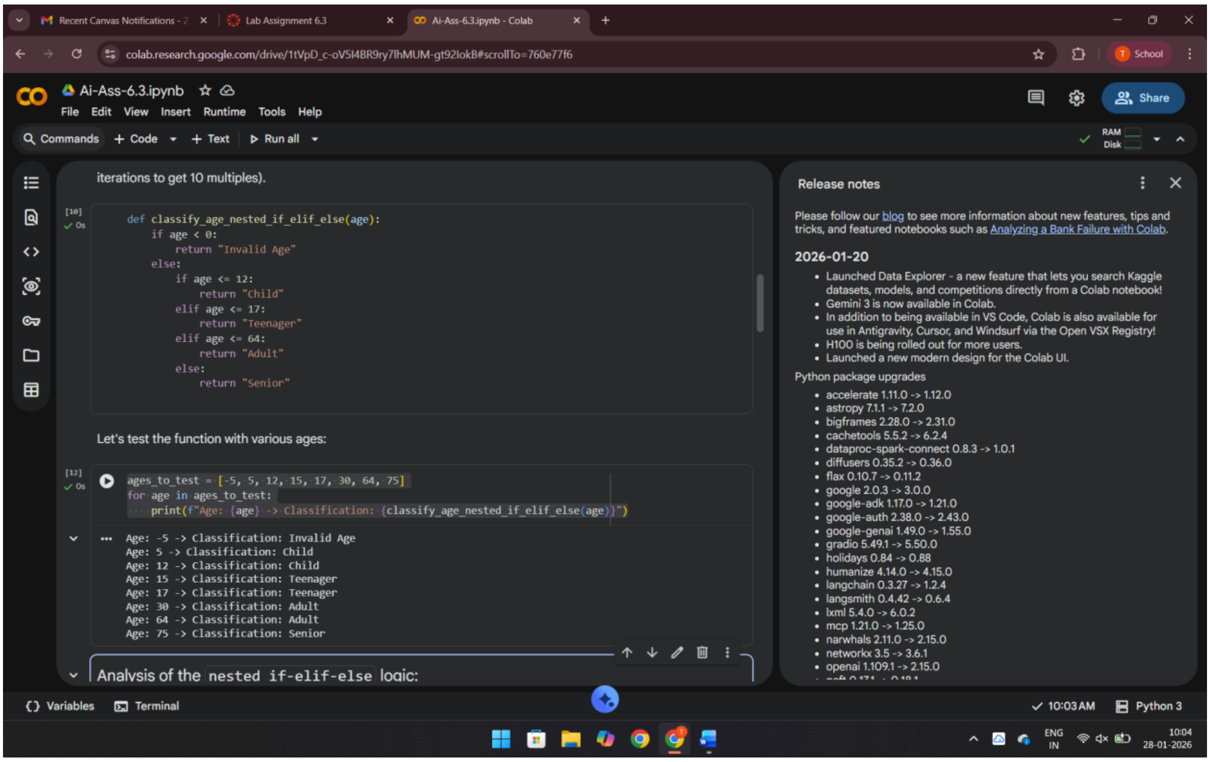
Code: def classify_age_nested_if_elif_else(age):

```
   if age < 0:
      return "Invalid Age"
   else:
      if age <= 12:
         return "Child"
      elif age <= 17:
         return "Teenager"
      elif age <= 64:
         return "Adult"
      else:
         return "Senior"
```

Output: ages_to_test = [-5, 5, 12, 15, 17, 30, 64, 75]

```
for age in ages_to_test:
   print(f"Age: {age} -> Classification: {classify_age_nested_if_elif_else(age)}")
```

1. Explanation: Nested if-elif-else: Uses an outer if-else for validity (negative age) and an inner if-elif-else for detailed classification (Child, Teenager, Adult, Senior).
2. Simplified if-elif-else: A flat structure where conditions are checked sequentially. It's generally more readable for ordered range checks.
3. Dictionary-based logic: Handles invalid age first, then iterates through a list of (limit, group) tuples to find the correct classification, defaulting to 'Senior' if no limit is met. This method is more adaptable for discrete categories or when ranges are defined externally.



Task Description #4: For and While Loops (Sum of First n Numbers)

Scenario

You need to calculate the sum of the first n natural numbers.

Task

• Use AI assistance to generate a sum_to_n() function using a for loop.

• Analyze the generated code.

• Ask the AI to suggest an alternative implementation using a while loop or a mathematical formula.

Expected Output #4

• Python function to compute the sum of first n numbers.

• Correct output for sample inputs.

• Explanation and comparison of different approaches.

Code:
```
def print_multiples_while(number):
    print(f"First 10 multiples of {number} (using while loop):")
    count = 1  # Initialize a counter
    while count <= 10: # Loop as long as count is 10 or less
```

```
    print(number * count)
    count += 1 # Increment the counter
```

Output: print_multiples_while(7)

First 10 multiples of 7 (using while loop):

7

14

21

28

35

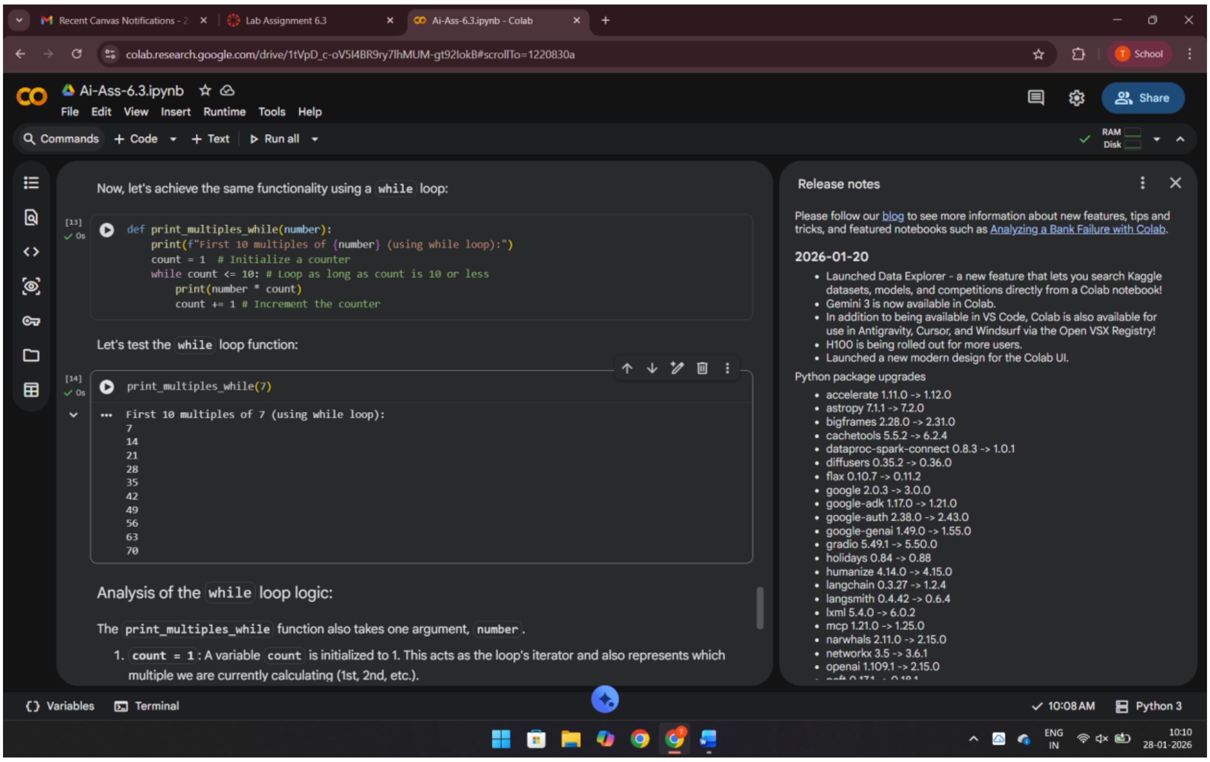
42

49

56

63

70



Explanation: The three sum_to_n functions demonstrate different ways to sum numbers up to n. The for loop and while loop implementations iterate from 1 to n, incrementally adding to a total. The for loop manages iteration automatically, while the while loop requires manual counter management. The mathematical formula n * (n + 1) // 2 provides the most efficient solution by directly calculating the sum without iteration.

Task Description #5: Classes (Bank Account Class)

Scenario

You are designing a basic banking application.

Task

• Use AI tools to generate a Bank Account class with methods such as deposit(), withdraw(),

and check_balance().

• Analyze the AI-generated class structure and logic.

• Add meaningful comments and explain the working of the code.

Expected Output #5

• Complete Python Bank Account class.

• Demonstration of deposit and withdrawal operations with updated balance.

• Well-commented code with a clear explanation.

Code:
```
class BankAccount:
    def __init__(self, account_holder_name, initial_balance=0):
        """
        Initializes a new BankAccount instance.

        Args:
            account_holder_name (str): The name of the account holder.
            initial_balance (float): The starting balance of the account. Defaults to 0.
        """
        self.account_holder_name = account_holder_name
        # Ensure initial balance is not negative
        if initial_balance >= 0:
```

```python
            self.balance = initial_balance
        else:
            print("Initial balance cannot be negative. Setting balance to 0.")
            self.balance = 0

    def deposit(self, amount):
        """
        Deposits a specified amount into the account.

        Args:
            amount (float): The amount to deposit.
        """
        if amount > 0:
            self.balance += amount
            print(f"Deposit of ${amount:.2f} successful. New balance: ${self.balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        """
        Withdraws a specified amount from the account.

        Args:
            amount (float): The amount to withdraw.
        """
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
```

```
            print("Insufficient funds.")
        else:
            self.balance -= amount
            print(f"Withdrawal of ${amount:.2f} successful. New balance: ${self.balance:.2f}")

    def check_balance(self):
        """
        Displays the current balance of the account.
        """
        print(f"Account Holder: {self.account_holder_name}")
        print(f"Current Balance: ${self.balance:.2f}")

# --- Example Usage ---
print("\n--- Creating Account --- ")
account1 = BankAccount("John Doe", 1000.00)
account1.check_balance()

print("\n--- Performing Transactions ---")
account1.deposit(500.50)
account1.withdraw(200.00)
account1.withdraw(1500.00) # Attempt to overdraw
account1.deposit(-100.00) # Attempt invalid deposit

print("\n--- Final Balance ---")
account1.check_balance()

account2 = BankAccount("Jane Smith")
account2.deposit(250.00)
```

account2.check_balance()

Ouput: --- Creating Account ---

Account Holder: John Doe

Current Balance: $1000.00

--- Performing Transactions ---

Deposit of $500.50 successful. New balance: $1500.50

Withdrawal of $200.00 successful. New balance: $1300.50

Insufficient funds.

Deposit amount must be positive.

--- Final Balance ---

Account Holder: John Doe

Current Balance: $1300.50

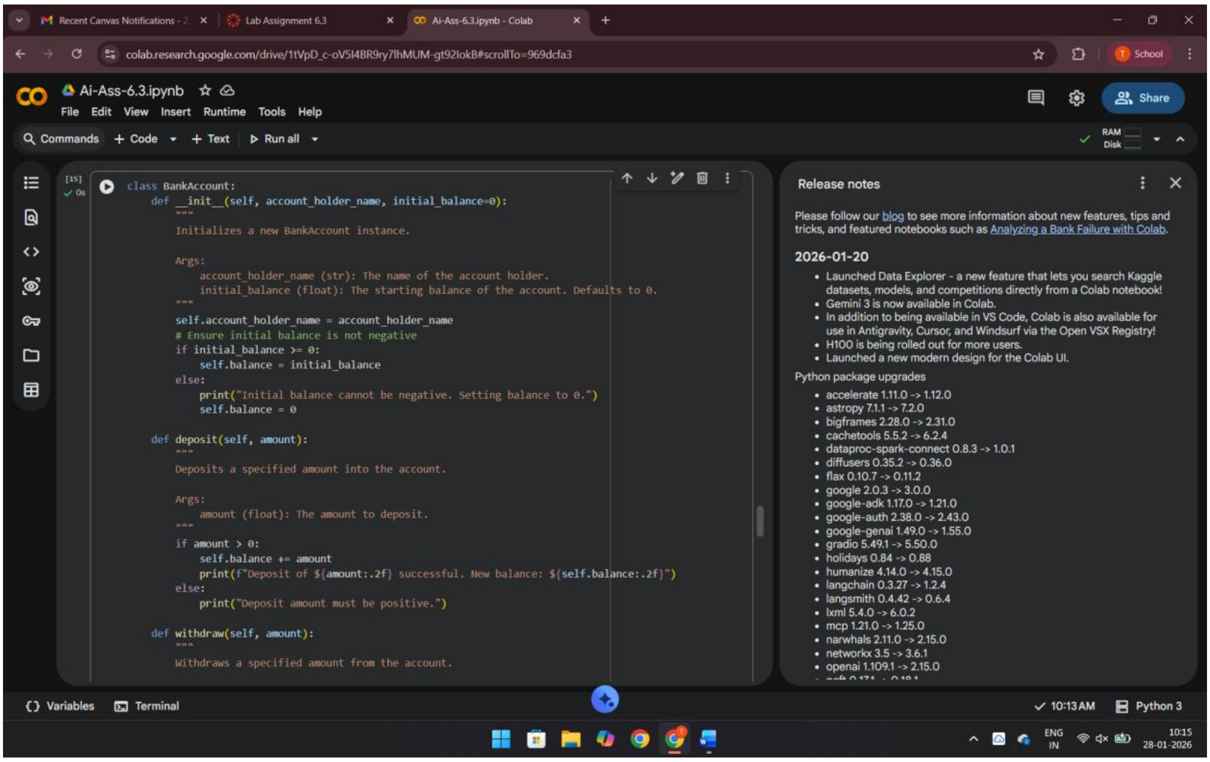
Deposit of $250.00 successful. New balance: $250.00

Account Holder: Jane Smith

Current Balance: $250.00

Explanation:

The BankAccount class models a bank account with an account holder name and balance. It provides methods for deposit (adds money, validates positive amounts), withdraw (removes money, validates positive amounts and sufficient funds), and check_balance (displays account information). It also includes initial balance validation.



Ai-Ass-6.3.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
class BankAccount:
    def __init__(self, account_holder_name, initial_balance=0):
        """
        Initializes a new BankAccount instance.

        Args:
            account_holder_name (str): The name of the account holder.
            initial_balance (float): The starting balance of the account. Defaults to 0.
        """
        self.account_holder_name = account_holder_name
        # Ensure initial balance is not negative
        if initial_balance >= 0:
            self.balance = initial_balance
        else:
            print("Initial balance cannot be negative. Setting balance to 0.")
            self.balance = 0

    def deposit(self, amount):
        """
        Deposits a specified amount into the account.

        Args:
            amount (float): The amount to deposit.
        """
        if amount > 0:
            self.balance += amount
            print(f"Deposit of ${amount:.2f} successful. New balance: ${self.balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        """
        Withdraws a specified amount from the account.
Release notes
Please follow our blog to see more information about new features, tips and tricks, and featured notebooks such as Analyzing a Bank Failure with Colab.
2026-01-20
Launched Data Explorer - a new feature that lets you search Kaggle datasets, models, and competitions directly from a Colab notebook!
Gemini 3 is now available in Colab.
In addition to being available in VS Code, Colab is also available for use in Antigravity, Cursor, and Windsurf via the Open VSX Registry!
H100 is being rolled out for more users.
Launched a new modern design for the Colab UI.
Python package upgrades
accelerate 1.11.0 -> 1.12.0
astropy 7.1.1 -> 7.2.0
bigframes 2.28.0 -> 2.31.0
cachetools 5.5.2 -> 6.2.4
dataproc-spark-connect 0.8.3 -> 1.0.1
diffusers 0.35.2 -> 0.36.0
flax 0.10.7 -> 0.11.2
google 2.0.3 -> 3.0.0
google-adk 1.17.0 -> 1.21.0
google-auth 2.38.0 -> 2.43.0
google-genai 1.49.0 -> 1.55.0
gradio 5.49.1 -> 5.50.0
holidays 0.84 -> 0.88
humanize 4.14.0 -> 4.15.0
langchain 0.3.27 -> 1.2.4
langsmith 0.4.42 -> 0.6.4
lxml 5.4.0 -> 6.0.2
mcp 1.21.0 -> 1.25.0
narwhals 2.11.0 -> 2.15.0
networkx 3.5 -> 3.6.1
openai 1.109.1 -> 2.15.0
Variables
Terminal
10:13 AM
Python 3