SR UNIVERSITY

Program	:B.tech(CSE)
Specialization	:AIML
Course Title	:AI Assisted Coding
Course Code	:24CS002PC215
Semester	:3rd semester
Academic Session	:2025-2026
Name of Student	:Kurapati Akshitha
Enrollment No.	:2403A52021
Batch No.	:02
Date	:16/09/2025

#LAB ASSIGNMENT

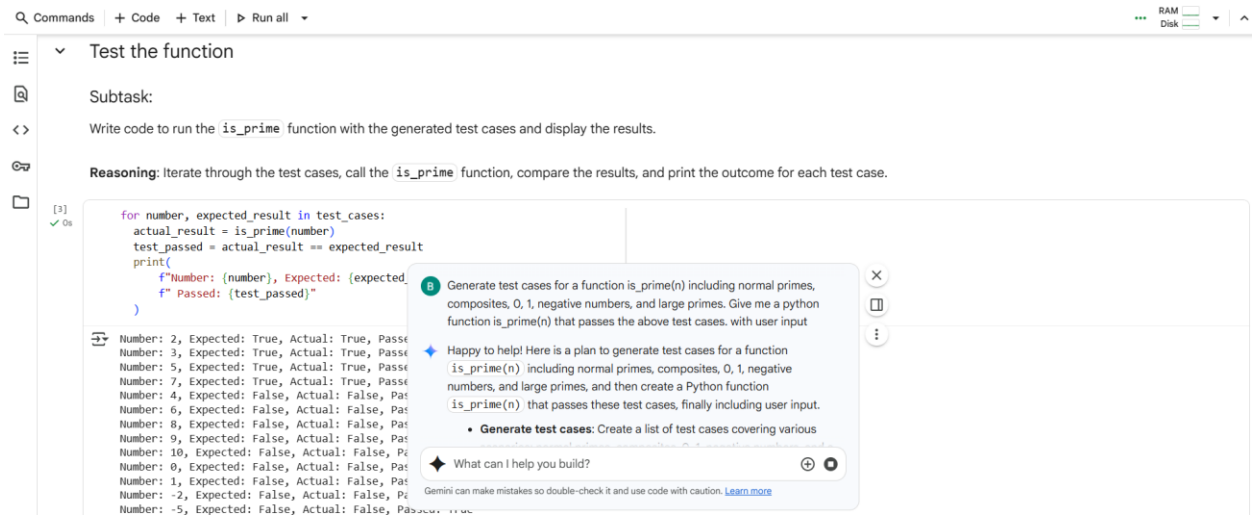
#TASK DESCRIPTION-1:

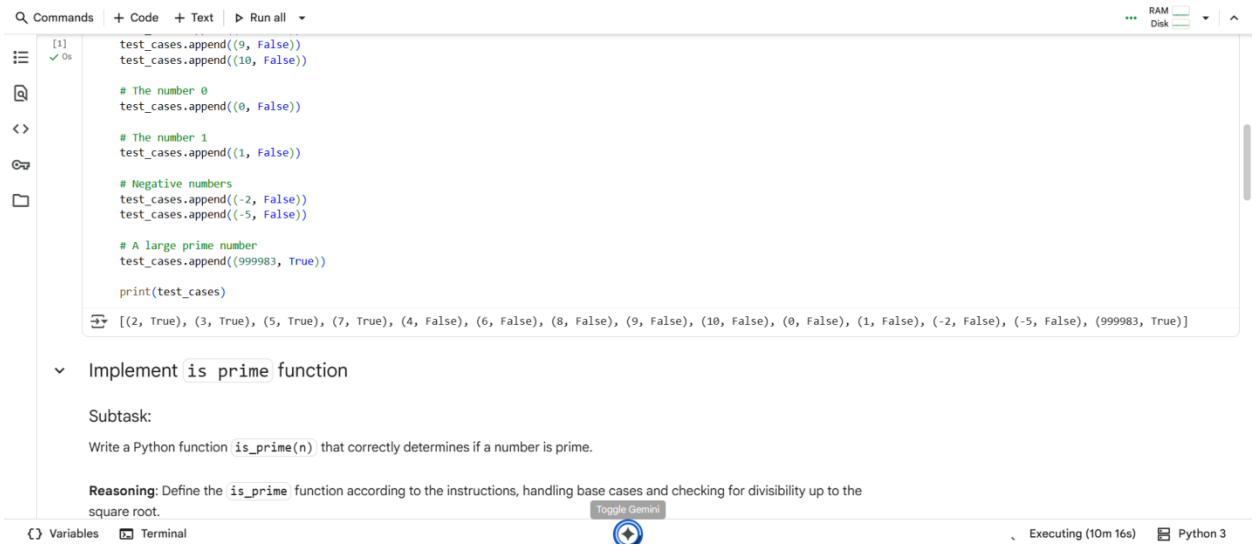
Use AI to generate test cases for a function is_prime(n) and then implement the function.

#PROMPT:

Generate test cases for a function is_prime(n) including normal primes, composites, 0, 1, negative numbers, and large primes. Give me a python function is_prime(n) that passes the above test cases. with user input.

#QUESTION:





#CODE with OUTPUT:

```
test_cases.append((9, False))
test_cases.append((10, False))

# The number 0
test_cases.append((0, False))

# The number 1
test_cases.append((1, False))

# Negative numbers
test_cases.append((-2, False))
test_cases.append((-5, False))

# A large prime number
test_cases.append((999983, True))

print(test_cases)
```

```
[(2, True), (3, True), (5, True), (7, True), (4, False), (6, False), (8, False), (9, False), (10, False), (0, False), (1, False), (-2, False), (-5, False), (999983, True)]
```

Implement `is prime` function

Subtask:

Write a Python function `is_prime(n)` that correctly determines if a number is prime.

Reasoning: Define the `is_prime` function according to the instructions, handling base cases and checking for divisibility up to the square root.

```
if n <= 1:
  return False
if n == 2:
  return True
if n % 2 == 0:
  return False
for i in range(3, int(math.sqrt(n)) + 1, 2):
  if n % i == 0:
    return False
return True
```

Test the function

Subtask:

Write code to run the `is_prime` function with the generated test cases and display the results.

Reasoning: Iterate through the test cases, call the `is_prime` function, compare the results, and print the outcome for each test case.

```
for number, expected_result in test_cases:
  actual_result = is_prime(number)
  test_passed = actual_result == expected_result
  print(
      f"Number: {number}, Expected: {expected_result}, Actual: {actual_result},"
      f" Passed: {test_passed}"
  )
```

Reasoning: The subtask is to allow the user to input a number and check if it's prime. This involves prompting for input, converting it to an integer while handling errors, calling the `is_prime` function, and printing the result. All these steps can be combined into a single code block.

```
while True:
    try:
        user_input = input("Enter a number to check if it's prime (or type 'quit' to exit): ")
        if user_input.lower() == 'quit':
            break
        number_to_check = int(user_input)
        if is_prime(number_to_check):
            print(f"{number_to_check} is a prime number.")
        else:
            print(f"{number_to_check} is not a prime number.")
    except ValueError:
        print("Invalid input. Please enter an integer.")
```

```
2 is a prime number.
7 is a prime number.
5 is a prime number.
18890 is not a prime number.
90 is not a prime number.
97 is a prime number.
57 is not a prime number.
Enter a number to check if it's prime (or type 'quit' to exit):
```

#EXPLANATION:

This code block implements a loop that continuously prompts the user to enter a number to check if it's prime. Here's a breakdown:

- `while True:`: This creates an infinite loop, allowing the user to check multiple numbers without restarting the code.
- `try...except ValueError:`: This block handles potential errors. It tries to execute the code within the `try` block. If a `ValueError` occurs (which happens if the user enters something that cannot be converted to an integer), the code within the `except` block is executed.
- `user_input = input(...)`: This line prompts the user to enter input and stores it in the `user_input` variable.
- `if user_input.lower() == 'quit': break`: This checks if the user typed 'quit' (case-insensitive). If so, the `break` statement exits the `while` loop.
- `number_to_check = int(user_input)`: This line converts the user's input into an integer. If the input is not a valid integer, a `ValueError` is raised, and the `except` block is executed.
- `if is_prime(number_to_check): ... else: ...`: This calls the `is_prime` function (defined earlier) to check if the entered number is prime. Based on the function's return value (True or False), it prints a message indicating whether the number is prime or not.
- `print("Invalid input. Please enter an integer.")`: This message is printed if a `ValueError` occurs, guiding the user to enter valid input.

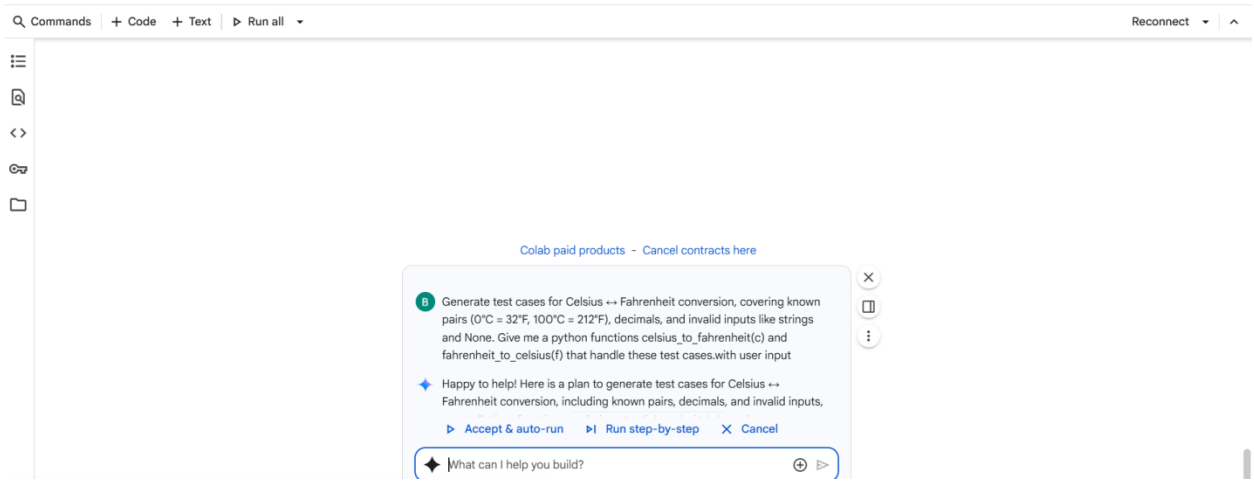
#TASK DESCRIPTION-2:

Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f).

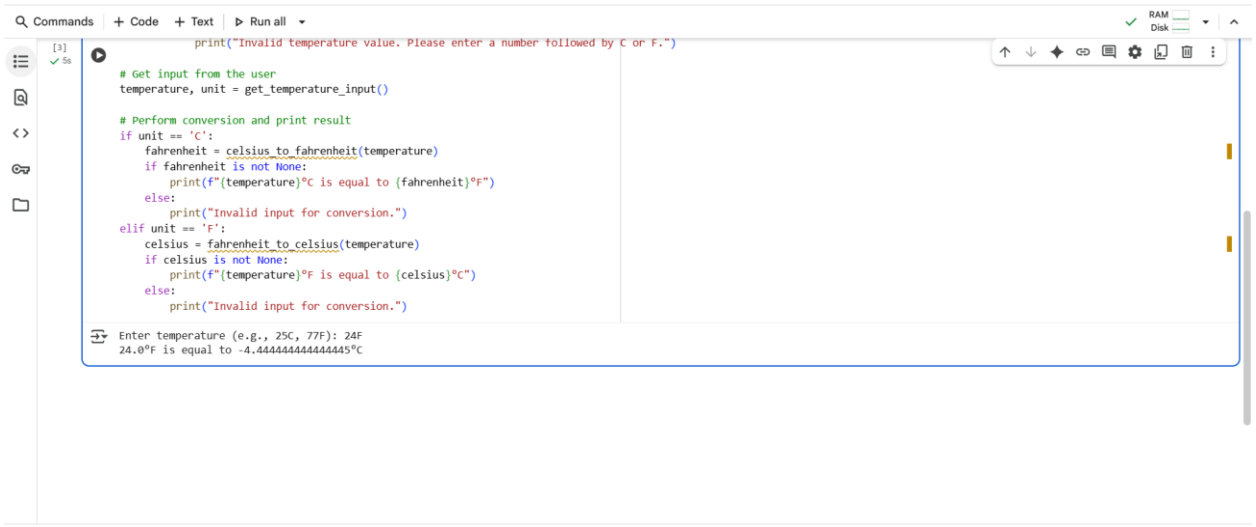
#PROMPT:

Generate test cases for Celsius ↔ Fahrenheit conversion, covering known pairs (0°C = 32°F, 100°C = 212°F), decimals, and invalid inputs like strings and None. Give me a python functions celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f) that handle these test cases.
With user input.

#QUESTION:



#CODE with OUTPUT:



#EXPLANATION:

This code is designed to take a temperature input from the user, determine if it's in Celsius or Fahrenheit, and then convert it to the other unit using the `celsius_to_fahrenheit` or `fahrenheit_to_celsius` functions (which were defined in a previous cell).

Here's a breakdown:

1. **`def get_temperature_input():`**: This defines a function named `get_temperature_input`. This function is responsible for getting valid temperature input from the user.
2. **`while True:`**: This starts an infinite loop. The loop will continue until a valid input is received and the function returns a value.
3. **`temp_str = input("Enter temperature (e.g., 25C, 77F): ")`**: This line prompts the user to enter a temperature and stores their input as a string in the `temp_str` variable.
4. **`if not temp_str:`**: This checks if the user entered an empty string. If they did, it prints a message and the loop continues.
5. **`unit = temp_str[-1].upper()`**: This extracts the last character of the input string (`temp_str[-1]`) and converts it to uppercase (`.upper()`). This is expected to be the unit ('C' or 'F').
6. **`value_str = temp_str[:-1]`**: This extracts all characters except the last one from the input string (`temp_str[:-1]`). This is expected to be the numerical value of the temperature.
7. **`try...except ValueError:`**: This is a `try-except` block to handle potential errors.
    - **`value = float(value_str)`**: Inside the `try` block, it attempts to convert the extracted `value_str` into a floating-point number.
    - **`if unit == 'C': return value, 'C'`**: If the unit is 'C', it returns the numerical value and the unit 'C'.
    - **`elif unit == 'F': return value, 'F'`**: If the unit is 'F', it returns the numerical value and the unit 'F'.
    - **`else: print("Invalid unit...")`**: If the unit is not 'C' or 'F', it prints an error message.
    - **`except ValueError: print("Invalid temperature value...")`**: If the `float(value_str)` conversion fails (e.g., the user entered non-numeric characters before the unit), a `ValueError` occurs, and this block prints an error message.
8. **`temperature, unit = get_temperature_input()`**: This line calls the `get_temperature_input()` function and unpacks the returned tuple into the `temperature` and `unit` variables.
9. **`if unit == 'C': ... elif unit == 'F': ...`**: This checks the `unit` variable.
    - If the unit is 'C', it calls the `celsius_to_fahrenheit()` function to convert the `temperature`.
    - If the unit is 'F', it calls the `fahrenheit_to_celsius()` function to convert the `temperature`.
10. **`if fahrenheit is not None: ... else: ...`** and **`if celsius is not None: ... else: ...`**: These checks ensure that the conversion functions returned a valid number (not `None`, which would indicate invalid input to the conversion function). If the conversion was successful, it prints the result in a user-friendly format. Otherwise, it indicates invalid input for conversion.

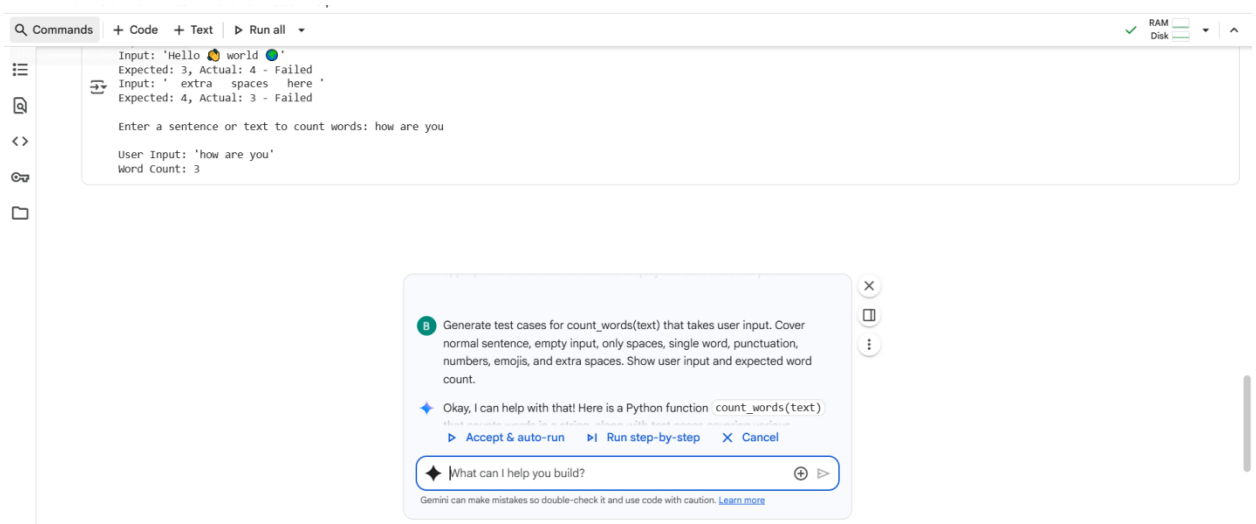
#TASK DESCRIPTION-3:

Use AI to write test cases for a function count_words(text) that returns the number of words in a sentence.

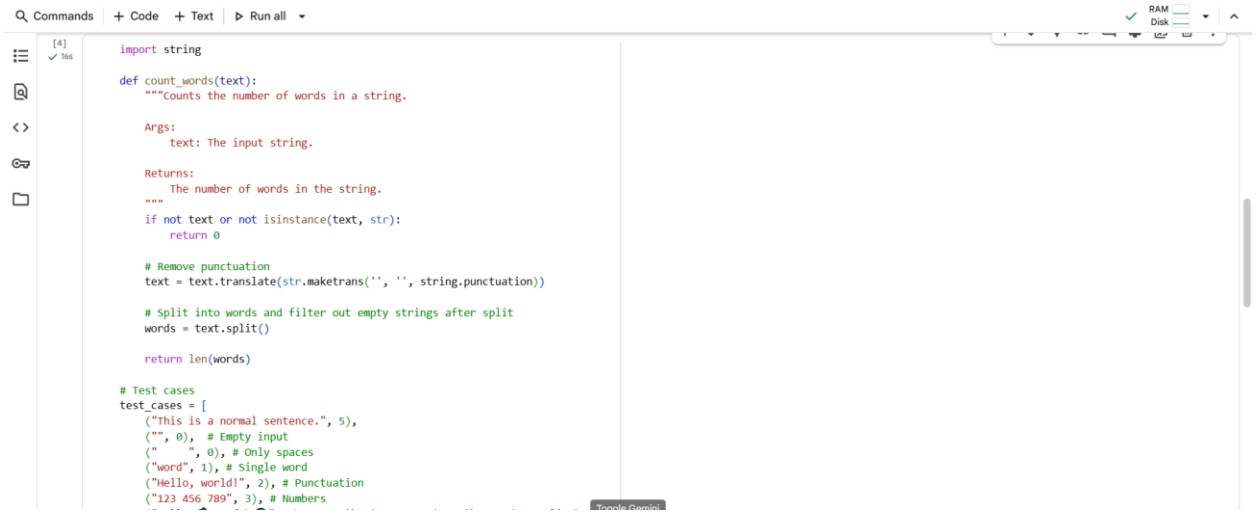
#PROMPT:

Generate test cases for `count_words(text)` that takes user input. Cover normal sentence, empty input, only spaces, single word, punctuation, numbers, emojis, and extra spaces. Show user input and expected word count.

#QUESTION:



#CODE with OUTPUT:



```
Commands + Code + Text ▷ Run all
[4]
    ("123 456 789", 3), # Numbers
    ("Hello 👋 world 🌎", 3), # Emojis (may vary depending on how split handles them)
    ("  extra   spaces   here ", 4) # Extra spaces
]

print("Word Count Tests:")
for text, expected_count in test_cases:
    actual_count = count_words(text)
    print(f"Input: '{text}'\nExpected: {expected_count}, Actual: {actual_count} - {'Passed' if actual_count == expected_count else 'Failed'}")

# Get user input and count words
user_input = input("\nEnter a sentence or text to count words: ")
user_word_count = count_words(user_input)
print(f"\nUser Input: '{user_input}'")
print(f"Word Count: {user_word_count}")

Word Count Tests:
Input: 'This is a normal sentence.'
Expected: 5, Actual: 5 - Passed
Input: ''
Expected: 0, Actual: 0 - Passed
Input: '     '
Expected: 0, Actual: 0 - Passed
Input: 'word'
Expected: 1, Actual: 1 - Passed
Input: 'Hello, world!'
Expected: 2, Actual: 2 - Passed
Input: '123 456 789'
Expected: 3, Actual: 3 - Passed
Input: 'Hello 👋 world 🌎'
Expected: 3, Actual: 4 - Failed
Input: '  extra   spaces   here '
Expected: 4, Actual: 3 - Failed

Enter a sentence or text to count words: how are you

User Input: 'how are you'
Word Count: 3
```

#TASK DESCRIPTION-4:

Generate test cases for a BankAccount class with:
Methods:
deposit(amount)
withdraw(amount)
check_balance()

Requirements:

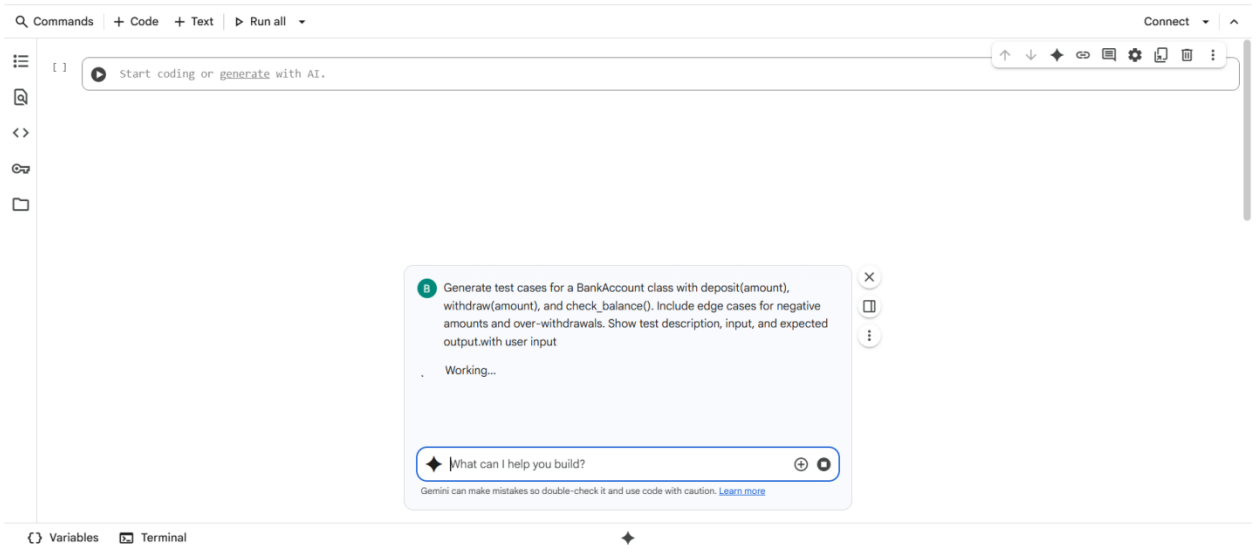
Negative deposits/withdrawals should raise an error.

Cannot withdraw more than balance.

#PROMPT:

Generate test cases for a BankAccount class with deposit(amount), withdraw(amount), and check_balance(). Include edge cases for negative amounts and over-withdrawals. Show test description, input, and expected output.

#QUESTION:



#CODE with OUTPUT:

#EXPLANATION:

- **`class BankAccount:`**: This line declares the beginning of a new class named `BankAccount`. Classes are blueprints for creating objects (in this case, bank accounts) that have specific properties (like a balance) and behaviors (like depositing or withdrawing).
- **`def __init__(self):`**: This is the constructor method. It's called automatically when you create a new `BankAccount` object. The `self` parameter refers to the instance of the class being created. Inside the constructor, `self.balance = 0` initializes the bank account's balance to zero when a new account is created.
- **`def deposit(self, amount):`**: This method handles depositing money into the account.
  - `if amount > 0:`: It checks if the deposit `amount` is positive.
  - `self.balance += amount`: If the amount is positive, it adds the `amount` to the current `self.balance`.
  - `print(f"Deposited: {amount}. New balance: {self.balance}")`: It prints a confirmation message showing the deposited amount and the new balance.
  - `else: print("Invalid deposit amount. Amount must be positive.")`: If the amount is not positive, it prints an error message.
- **`def withdraw(self, amount):`**: This method handles withdrawing money from the account.
  - `if amount > 0:`: It checks if the withdrawal `amount` is positive.
  - `if self.balance >= amount:`: If the amount is positive, it then checks if there are sufficient funds in the account (`self.balance` is greater than or equal to the `amount` to be withdrawn).
  - `self.balance -= amount`: If there are sufficient funds, it subtracts the `amount` from the current `self.balance`.
  - `print(f"Withdrew: {amount}. New balance: {self.balance}")`: It prints a confirmation message showing the withdrawn amount and the new balance.
  - `else: print("Insufficient funds.")`: If there are insufficient funds, it prints an "Insufficient funds" message.
  - `else: print("Invalid withdrawal amount. Amount must be positive.")`: If the withdrawal amount is not positive, it prints an error message.
- **`def check_balance(self):`**: This method simply returns the current balance of the account (`self.balance`).

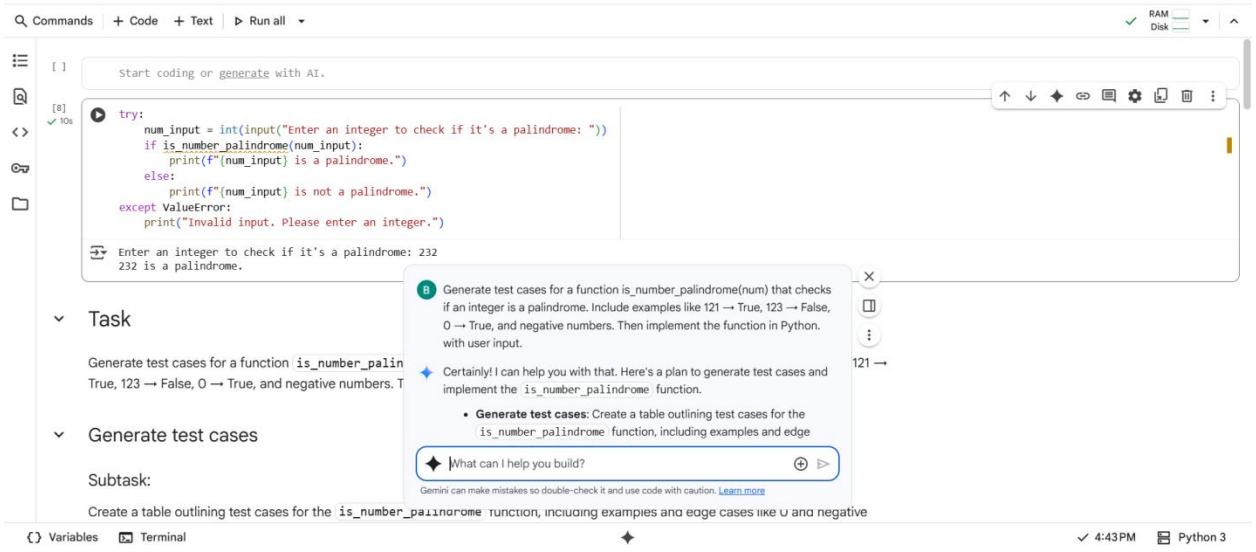
#TASK DESCRIPTION-5:

Generate test cases for is_number_palindrome(num), which checks if an integer reads the same backward.

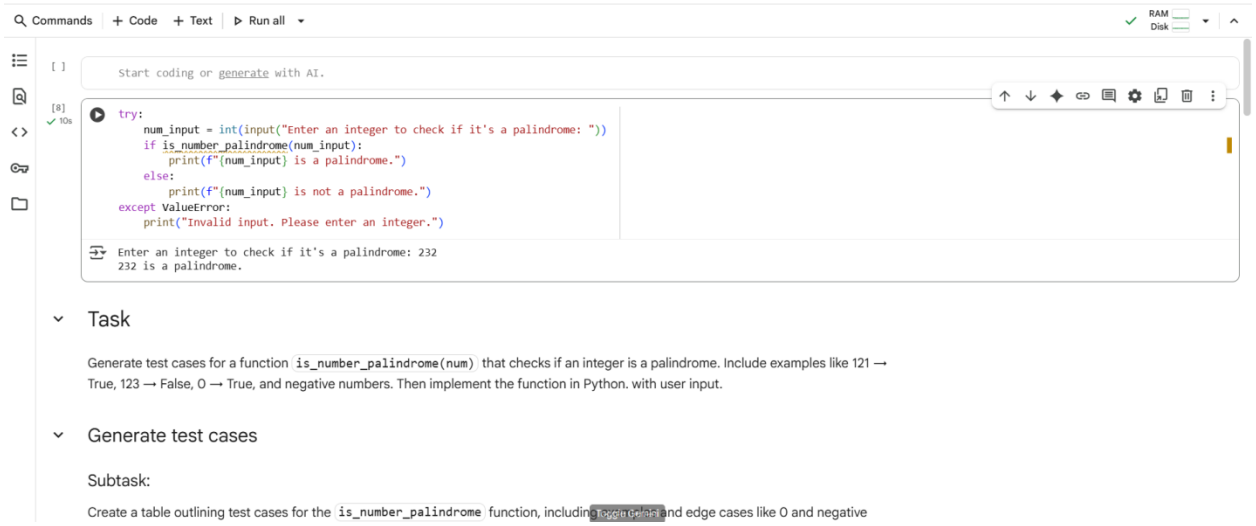
#PROMPT:

Generate test cases for a function is_number_palindrome(num) that checks if an integer is a palindrome. Include examples like 121 → True, 123 → False, 0 → True, and negative numbers. Then implement the function in Python. with user input.

#QUESTION:



#CODE with OUTPUT:



Commands | + Code | + Text | Run all

RAM Disk

Subtask:

Create a table outlining test cases for the `is_number_palindrome` function, including examples and edge cases like 0 and negative numbers.

Reasoning: Create a list of dictionaries representing test cases for the `is_number_palindrome` function, including various examples and edge cases, then display the test cases using a pandas DataFrame for clear presentation as requested in the instructions.

[5] ✓ 0s

```
import pandas as pd

test_cases = [
    {'input': 121, 'expected_output': True},
    {'input': 123, 'expected_output': False},
    {'input': 0, 'expected_output': True},
    {'input': -121, 'expected_output': False}, # Negative numbers are not palindromes
    {'input': 5, 'expected_output': True},
    {'input': 11, 'expected_output': True},
    {'input': 10, 'expected_output': False},
    {'input': 1221, 'expected_output': True},
    {'input': 12321, 'expected_output': True},
    {'input': 12345, 'expected_output': False},
    {'input': -101, 'expected_output': False} # Another negative test case
]

test_cases_df = pd.DataFrame(test_cases)
display(test_cases_df)
```

Toggle Gemini

Commands | + Code | + Text | Run all

RAM Disk

	input	expected_output
0	121	True
1	123	False
2	0	True
3	-121	False
4	5	True
5	11	True
6	10	False
7	1221	True
8	12321	True
9	12345	False
10	-101	False

Next steps: Generate code with test_cases_df | New interactive sheet

Implement function

Subtask:

Write the Python code for the `is_number_palindrome(num)` function.

Toggle Gemini

Variables | Terminal

4:43 PM | Python 3

Type here to search

04:46 PM

Untitled28.ipynb

File Edit View Insert Runtime Tools Help

Share | Gemini

Commands | + Code | + Text | Run all

RAM Disk

[7] ✓ 0s

```
for index, row in test_cases_df.iterrows():
    input_num = row['input']
    expected_output = row['expected_output']
    actual_output = is_number_palindrome(input_num)

    print(f"Input: {input_num}, Expected: {expected_output}, Actual: {actual_output}")

    if actual_output == expected_output:
        print("Test case passed!")
    else:
        print("Test case failed!")
```

```
Input: 121, Expected: True, Actual: True
Test case passed!
Input: 123, Expected: False, Actual: False
Test case passed!
Input: 0, Expected: True, Actual: True
Test case passed!
Input: -121, Expected: False, Actual: False
Test case passed!
Input: 5, Expected: True, Actual: True
Test case passed!
Input: 11, Expected: True, Actual: True
Test case passed!
Input: 10, Expected: False, Actual: False
Test case passed!
Input: 1221, Expected: True, Actual: True
Test case passed!
Input: 12321, Expected: True, Actual: True
Test case passed!
Input: 12345, Expected: False, Actual: False
Test case passed!
Input: -101, Expected: False, Actual: False
```

Toggle Gemini

#EXPLANATION:

- **`for index, row in test_cases_df.iterrows():`**: This loop iterates through each row of the `test_cases_df` DataFrame. In each iteration:
  - `index` gets the index of the current row.
  - `row` gets a Series object representing the data in the current row.
- **`input_num = row['input']`**: This line extracts the value from the 'input' column of the current row and assigns it to the variable `input_num`. This is the number that will be passed to the `is_number_palindrome` function.
- **`expected_output = row['expected_output']`**: This line extracts the value from the 'expected_output' column of the current row and assigns it to the variable `expected_output`. This is the expected result for the given input, according to our test cases.
- **`actual_output = is_number_palindrome(input_num)`**: This line calls the `is_number_palindrome` function with the `input_num` and stores the returned value (either `True` or `False`) in the `actual_output` variable.
- **`print(f"Input: {input_num}, Expected: {expected_output}, Actual: {actual_output}")`**: This line prints the input number, the expected output, and the actual output from the function call for the current test case. This makes it easy to see the results of each test.
- **`if actual_output == expected_output:`**: This conditional statement compares the `actual_output` from the function with the `expected_output` from the test case.
  - **`print("Test case passed!")`**: If the actual output matches the expected output, it prints "Test case passed!".
  - **`else: print("Test case failed!")`**: If the actual output does not match the expected output, it prints "Test case failed!".

THANK YOU