Lab Assignment – 8.2

Course: AI - Assisted Coding

Name :P.Srikar

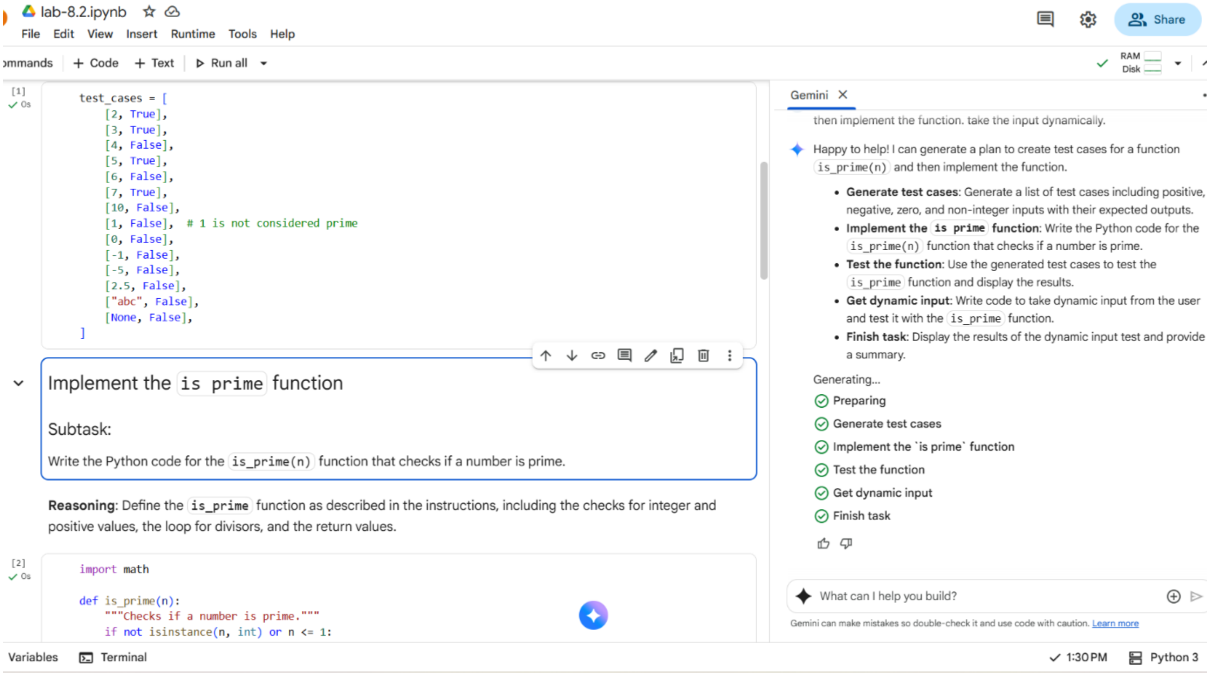
Htno :2403A52002

Task-1:

Prompt:

write a python code to generate test cases for a function is_prime(n) and then implement the function. take the input dynamically.

Code&Output:


lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
ommands + Code + Text ▷ Run all
Share
RAM Disk
Gemini ✕
Variables Terminal
✓ 1:30 PM Python 3

```
[1]
test_cases = [
    [2, True],
    [3, True],
    [4, False],
    [5, True],
    [6, False],
    [7, True],
    [10, False],
    [1, False],  # 1 is not considered prime
    [0, False],
    [-1, False],
    [-5, False],
    [2.5, False],
    ["abc", False],
    [None, False],
]
```

Implement the `is prime` function

Subtask:

Write the Python code for the `is_prime(n)` function that checks if a number is prime.

Reasoning: Define the `is_prime` function as described in the instructions, including the checks for integer and positive values, the loop for divisors, and the return values.

```
[2]
import math

def is_prime(n):
    """Checks if a number is prime."""
    if not isinstance(n, int) or n <= 1:
```

then implement the function. take the input dynamically.

Happy to help! I can generate a plan to create test cases for a function `is_prime(n)` and then implement the function.

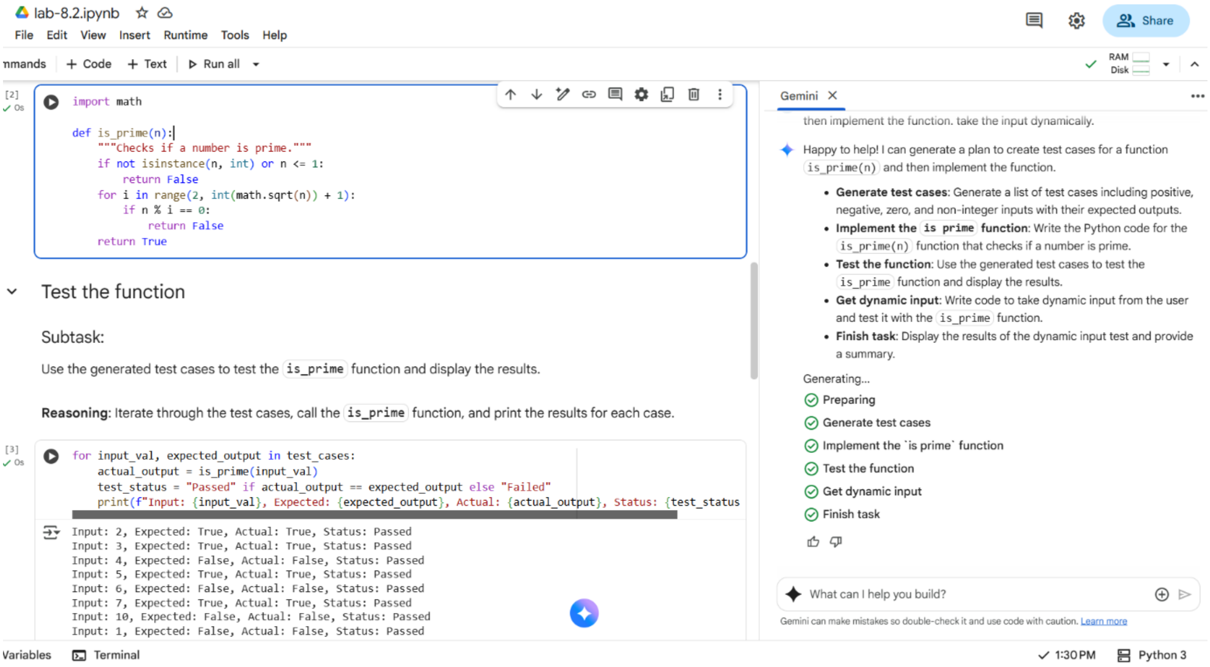
- **Generate test cases**: Generate a list of test cases including positive, negative, zero, and non-integer inputs with their expected outputs.
- **Implement the `is prime` function**: Write the Python code for the `is_prime(n)` function that checks if a number is prime.
- **Test the function**: Use the generated test cases to test the `is_prime` function and display the results.
- **Get dynamic input**: Write code to take dynamic input from the user and test it with the `is_prime` function.
- **Finish task**: Display the results of the dynamic input test and provide a summary.

Generating...

✓ Preparing
✓ Generate test cases
✓ Implement the `is prime` function
✓ Test the function
✓ Get dynamic input
✓ Finish task

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
import math

def is_prime(n):
    """Checks if a number is prime."""
    if not isinstance(n, int) or n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True
```

Test the function

Subtask:

Use the generated test cases to test the `is_prime` function and display the results.

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

```
for input_val, expected_output in test_cases:
    actual_output = is_prime(input_val)
    test_status = "Passed" if actual_output == expected_output else "Failed"
    print(f"Input: {input_val}, Expected: {expected_output}, Actual: {actual_output}, Status: {test_status
```

```
Input: 2, Expected: True, Actual: True, Status: Passed
Input: 3, Expected: True, Actual: True, Status: Passed
Input: 4, Expected: False, Actual: False, Status: Passed
Input: 5, Expected: True, Actual: True, Status: Passed
Input: 6, Expected: False, Actual: False, Status: Passed
Input: 7, Expected: True, Actual: True, Status: Passed
Input: 10, Expected: False, Actual: False, Status: Passed
Input: 1, Expected: False, Actual: False, Status: Passed
```

Gemini

then implement the function. take the input dynamically.

Happy to help! I can generate a plan to create test cases for a function `is_prime(n)` and then implement the function.

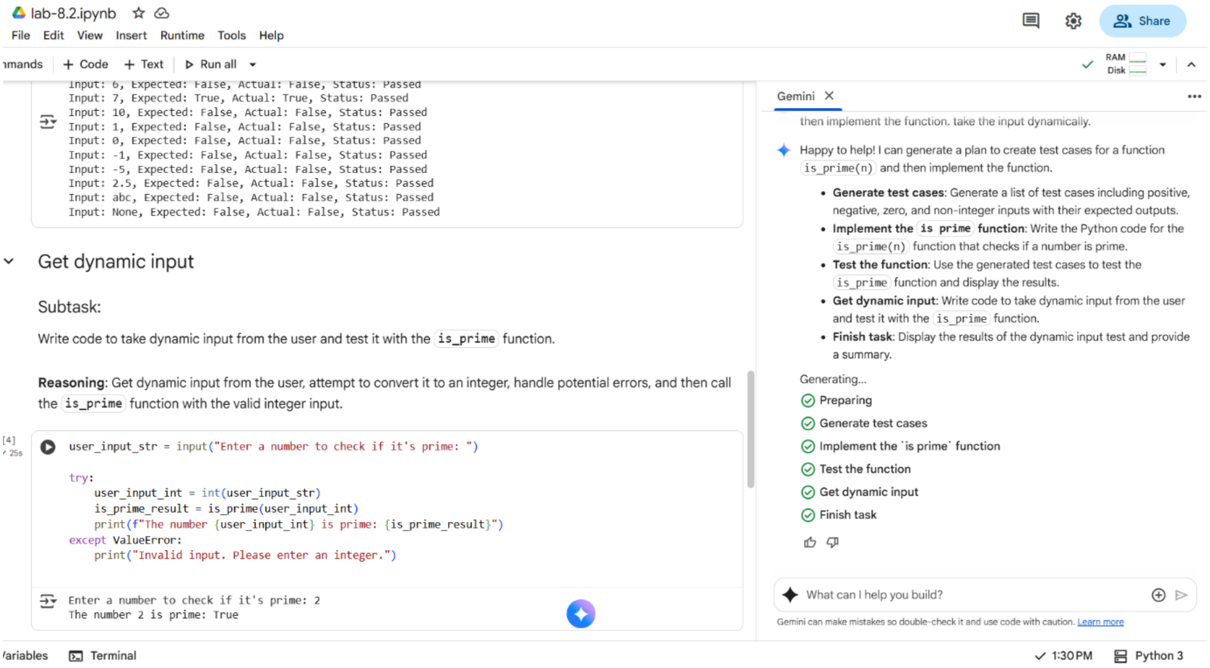
- **Generate test cases**: Generate a list of test cases including positive, negative, zero, and non-integer inputs with their expected outputs.
- **Implement the `is prime` function**: Write the Python code for the `is_prime(n)` function that checks if a number is prime.
- **Test the function**: Use the generated test cases to test the `is_prime` function and display the results.
- **Get dynamic input**: Write code to take dynamic input from the user and test it with the `is_prime` function.
- **Finish task**: Display the results of the dynamic input test and provide a summary.

Generating...

✓ Preparing
✓ Generate test cases
✓ Implement the `is prime` function
✓ Test the function
✓ Get dynamic input
✓ Finish task

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more


lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
Share
mmands + Code + Text Run all
RAM Disk
[4] 25s
Variables Terminal
1:30 PM Python 3

```
Input: 6, Expected: False, Actual: False, Status: Passed
Input: 7, Expected: True, Actual: True, Status: Passed
Input: 10, Expected: False, Actual: False, Status: Passed
Input: 1, Expected: False, Actual: False, Status: Passed
Input: 0, Expected: False, Actual: False, Status: Passed
Input: -1, Expected: False, Actual: False, Status: Passed
Input: -5, Expected: False, Actual: False, Status: Passed
Input: 2.5, Expected: False, Actual: False, Status: Passed
Input: abc, Expected: False, Actual: False, Status: Passed
Input: None, Expected: False, Actual: False, Status: Passed
```

Get dynamic input

Subtask:

Write code to take dynamic input from the user and test it with the `is_prime` function.

Reasoning: Get dynamic input from the user, attempt to convert it to an integer, handle potential errors, and then call the `is_prime` function with the valid integer input.

```
user_input_str = input("Enter a number to check if it's prime: ")

try:
    user_input_int = int(user_input_str)
    is_prime_result = is_prime(user_input_int)
    print(f"The number {user_input_int} is prime: {is_prime_result}")
except ValueError:
    print("Invalid input. Please enter an integer.")
```

```
Enter a number to check if it's prime: 2
The number 2 is prime: True
```

Gemini

then implement the function. take the input dynamically.

Happy to help! I can generate a plan to create test cases for a function `is_prime(n)` and then implement the function.

- **Generate test cases**: Generate a list of test cases including positive, negative, zero, and non-integer inputs with their expected outputs.
- **Implement the `is prime` function**: Write the Python code for the `is_prime(n)` function that checks if a number is prime.
- **Test the function**: Use the generated test cases to test the `is_prime` function and display the results.
- **Get dynamic input**: Write code to take dynamic input from the user and test it with the `is_prime` function.
- **Finish task**: Display the results of the dynamic input test and provide a summary.

Generating...

✓ Preparing
✓ Generate test cases
✓ Implement the `is prime` function
✓ Test the function
✓ Get dynamic input
✓ Finish task

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Observation:

The generated test cases cover a good range of inputs, including valid integers, edge cases like 0 and 1, negative numbers, and non-integer types.

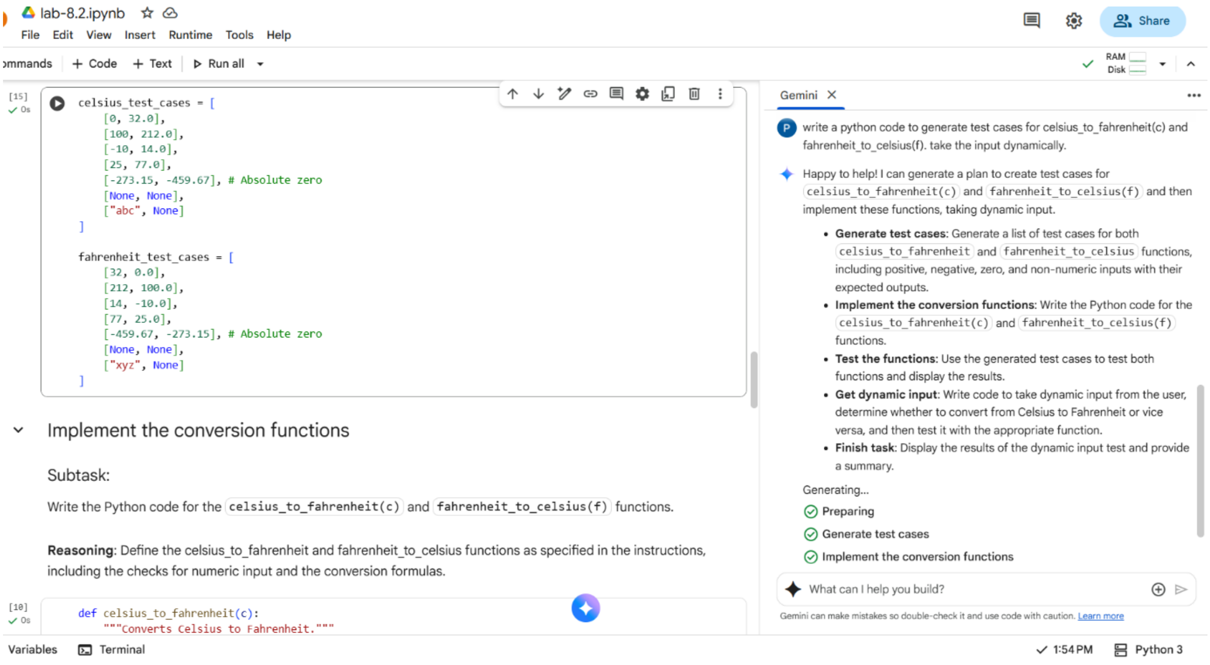
The is_prime function correctly handles these various inputs and efficiently determines primality for integers greater than 1.

The code successfully integrates dynamic user input and provides appropriate feedback, including error handling for invalid input types.

Task-2:

Prompt: write a python code to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). take the input dynamically.

Code&Output:



```
celsius_test_cases = [
    [0, 32.0],
    [100, 212.0],
    [-10, 14.0],
    [25, 77.0],
    [-273.15, -459.67], # Absolute zero
    [None, None],
    ["abc", None]
]

fahrenheit_test_cases = [
    [32, 0.0],
    [212, 100.0],
    [14, -10.0],
    [77, 25.0],
    [-459.67, -273.15], # Absolute zero
    [None, None],
    ["xyz", None]
]
```

Implement the conversion functions

Subtask:

Write the Python code for the `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` functions.

Reasoning: Define the celsius_to_fahrenheit and fahrenheit_to_celsius functions as specified in the instructions, including the checks for numeric input and the conversion formulas.

```
def celsius_to_fahrenheit(c):
    """Converts Celsius to Fahrenheit."""
```

write a python code to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). take the input dynamically.

Happy to help! I can generate a plan to create test cases for `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` and then implement these functions, taking dynamic input.

- **Generate test cases:** Generate a list of test cases for both `celsius_to_fahrenheit` and `fahrenheit_to_celsius` functions, including positive, negative, zero, and non-numeric inputs with their expected outputs.
- **Implement the conversion functions:** Write the Python code for the `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` functions.
- **Test the functions:** Use the generated test cases to test both functions and display the results.
- **Get dynamic input**: Write code to take dynamic input from the user, determine whether to convert from Celsius to Fahrenheit or vice versa, and then test it with the appropriate function.
- **Finish task**: Display the results of the dynamic input test and provide a summary.

Generating...

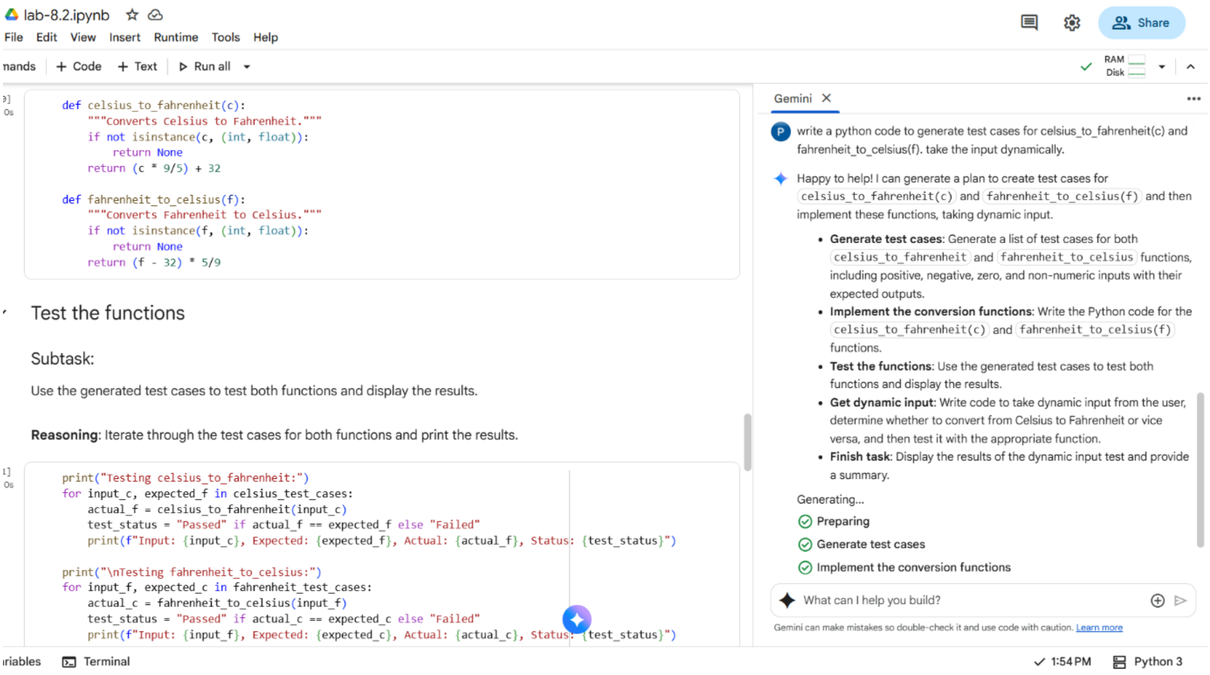
✓ Preparing

✓ Generate test cases

✓ Implement the conversion functions

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more


lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
nands + Code + Text ▷ Run all
Share
RAM
Disk
0s
Gemini ✕
0s
riables
Terminal
✓ 1:54 PM
Python 3

```
def celsius_to_fahrenheit(c):
    """Converts Celsius to Fahrenheit."""
    if not isinstance(c, (int, float)):
        return None
    return (c * 9/5) + 32

def fahrenheit_to_celsius(f):
    """Converts Fahrenheit to Celsius."""
    if not isinstance(f, (int, float)):
        return None
    return (f - 32) * 5/9
```

Test the functions

Subtask:

Use the generated test cases to test both functions and display the results.

Reasoning: Iterate through the test cases for both functions and print the results.

```
print("Testing celsius_to_fahrenheit:")
for input_c, expected_f in celsius_test_cases:
    actual_f = celsius_to_fahrenheit(input_c)
    test_status = "Passed" if actual_f == expected_f else "Failed"
    print(f"Input: {input_c}, Expected: {expected_f}, Actual: {actual_f}, Status: {test_status}")

print("\nTesting fahrenheit_to_celsius:")
for input_f, expected_c in fahrenheit_test_cases:
    actual_c = fahrenheit_to_celsius(input_f)
    test_status = "Passed" if actual_c == expected_c else "Failed"
    print(f"Input: {input_f}, Expected: {expected_c}, Actual: {actual_c}, Status: {test_status}")
```

write a python code to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). take the input dynamically.

Happy to help! I can generate a plan to create test cases for `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` and then implement these functions, taking dynamic input.

- **Generate test cases:** Generate a list of test cases for both `celsius_to_fahrenheit` and `fahrenheit_to_celsius` functions, including positive, negative, zero, and non-numeric inputs with their expected outputs.
- **Implement the conversion functions:** Write the Python code for the `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` functions.
- **Test the functions:** Use the generated test cases to test both functions and display the results.
- **Get dynamic input**: Write code to take dynamic input from the user, determine whether to convert from Celsius to Fahrenheit or vice versa, and then test it with the appropriate function.
- **Finish task**: Display the results of the dynamic input test and provide a summary.

Generating...

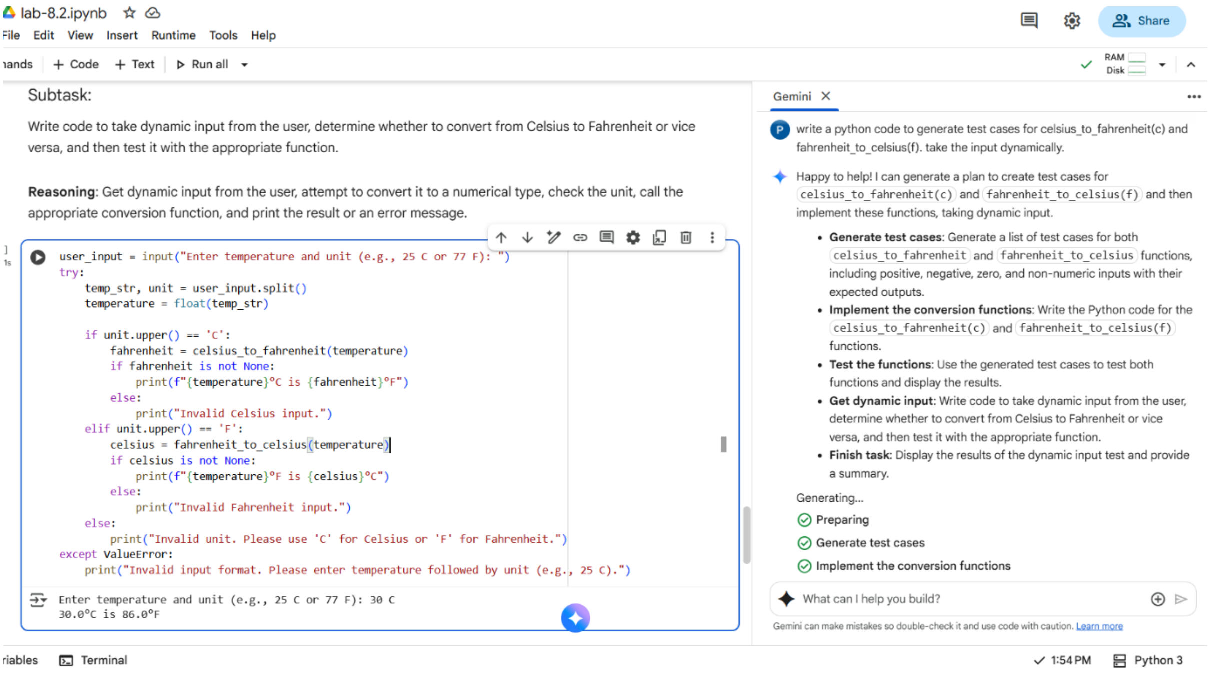
✓ Preparing

✓ Generate test cases

✓ Implement the conversion functions

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more


lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
ands + Code + Text ▷ Run all
Share
RAM
Disk
Gemini

Subtask:

Write code to take dynamic input from the user, determine whether to convert from Celsius to Fahrenheit or vice versa, and then test it with the appropriate function.

Reasoning: Get dynamic input from the user, attempt to convert it to a numerical type, check the unit, call the appropriate conversion function, and print the result or an error message.

```
user_input = input("Enter temperature and unit (e.g., 25 C or 77 F): ")
try:
    temp_str, unit = user_input.split()
    temperature = float(temp_str)

    if unit.upper() == 'C':
        fahrenheit = celsius_to_fahrenheit(temperature)
        if fahrenheit is not None:
            print(f"{temperature}°C is {fahrenheit}°F")
        else:
            print("Invalid Celsius input.")
    elif unit.upper() == 'F':
        celsius = fahrenheit_to_celsius(temperature)
        if celsius is not None:
            print(f"{temperature}°F is {celsius}°C")
        else:
            print("Invalid Fahrenheit input.")
    else:
        print("Invalid unit. Please use 'C' for Celsius or 'F' for Fahrenheit.")
except ValueError:
    print("Invalid input format. Please enter temperature followed by unit (e.g., 25 C).")
```

```
Enter temperature and unit (e.g., 25 C or 77 F): 30 C
30.0°C is 86.0°F
```

write a python code to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). take the input dynamically.

Happy to help! I can generate a plan to create test cases for `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` and then implement these functions, taking dynamic input.

- **Generate test cases**: Generate a list of test cases for both `celsius_to_fahrenheit` and `fahrenheit_to_celsius` functions, including positive, negative, zero, and non-numeric inputs with their expected outputs.
- **Implement the conversion functions**: Write the Python code for the `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` functions.
- **Test the functions**: Use the generated test cases to test both functions and display the results.
- **Get dynamic input**: Write code to take dynamic input from the user, determine whether to convert from Celsius to Fahrenheit or vice versa, and then test it with the appropriate function.
- **Finish task**: Display the results of the dynamic input test and provide a summary.

Generating...

✓ Preparing

✓ Generate test cases

✓ Implement the conversion functions

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Observation:

Test cases were successfully generated for both Celsius to Fahrenheit and Fahrenheit to Celsius conversions, covering a good range of inputs including edge cases like absolute zero and non-numeric types.

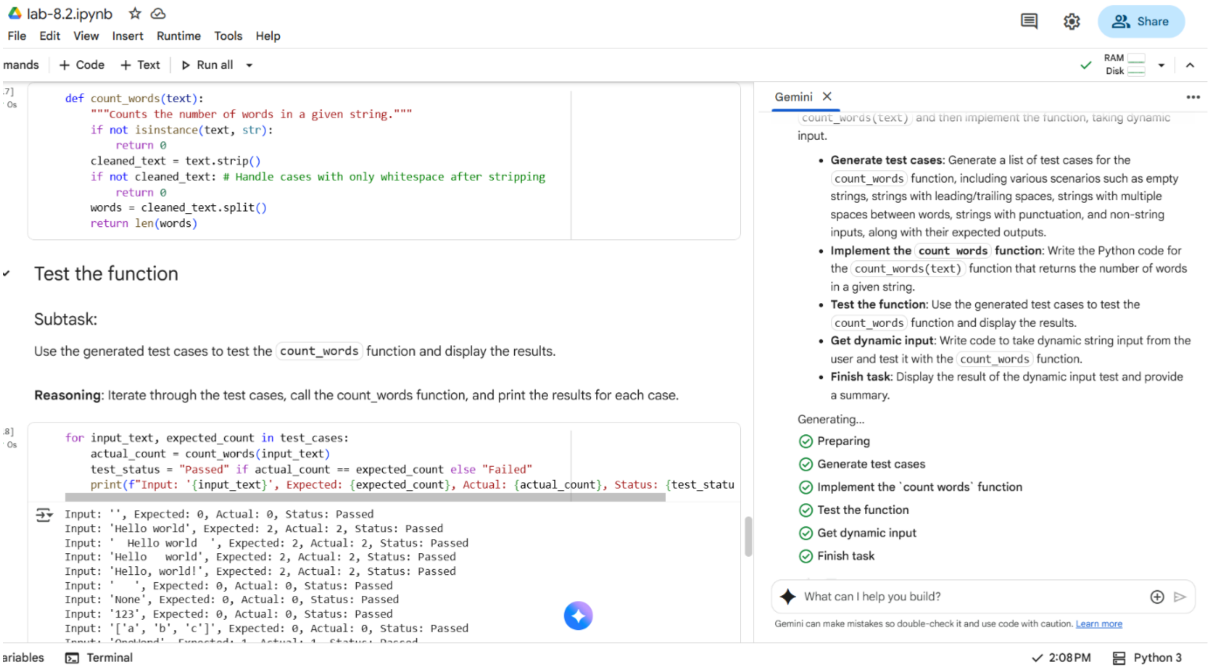
The dynamic input section correctly handles user input in the specified format, performs the appropriate conversion based on the unit provided, and includes error handling for invalid input formats or units.

Task-3:

Prompt:

write a python code to generate test cases for a function count_words(text) that returns the number of words in a sentence. take the input dynamically.

Code&Output:



```python
def count_words(text):
    """Counts the number of words in a given string."""
    if not isinstance(text, str):
        return 0
    cleaned_text = text.strip()
    if not cleaned_text: # Handle cases with only whitespace after stripping
        return 0
    words = cleaned_text.split()
    return len(words)
```

Test the function

Subtask:

Use the generated test cases to test the `count_words` function and display the results.

Reasoning: Iterate through the test cases, call the count_words function, and print the results for each case.

```python
for input_text, expected_count in test_cases:
    actual_count = count_words(input_text)
    test_status = "Passed" if actual_count == expected_count else "Failed"
    print(f"Input: '{input_text}', Expected: {expected_count}, Actual: {actual_count}, Status: {test_statu
```

```
Input: '', Expected: 0, Actual: 0, Status: Passed
Input: 'Hello world', Expected: 2, Actual: 2, Status: Passed
Input: '  Hello world  ', Expected: 2, Actual: 2, Status: Passed
Input: 'Hello   world', Expected: 2, Actual: 2, Status: Passed
Input: 'Hello, world!', Expected: 2, Actual: 2, Status: Passed
Input: '   ', Expected: 0, Actual: 0, Status: Passed
Input: 'None', Expected: 0, Actual: 0, Status: Passed
Input: '123', Expected: 0, Actual: 0, Status: Passed
Input: '['a', 'b', 'c']', Expected: 0, Actual: 0, Status: Passed
```

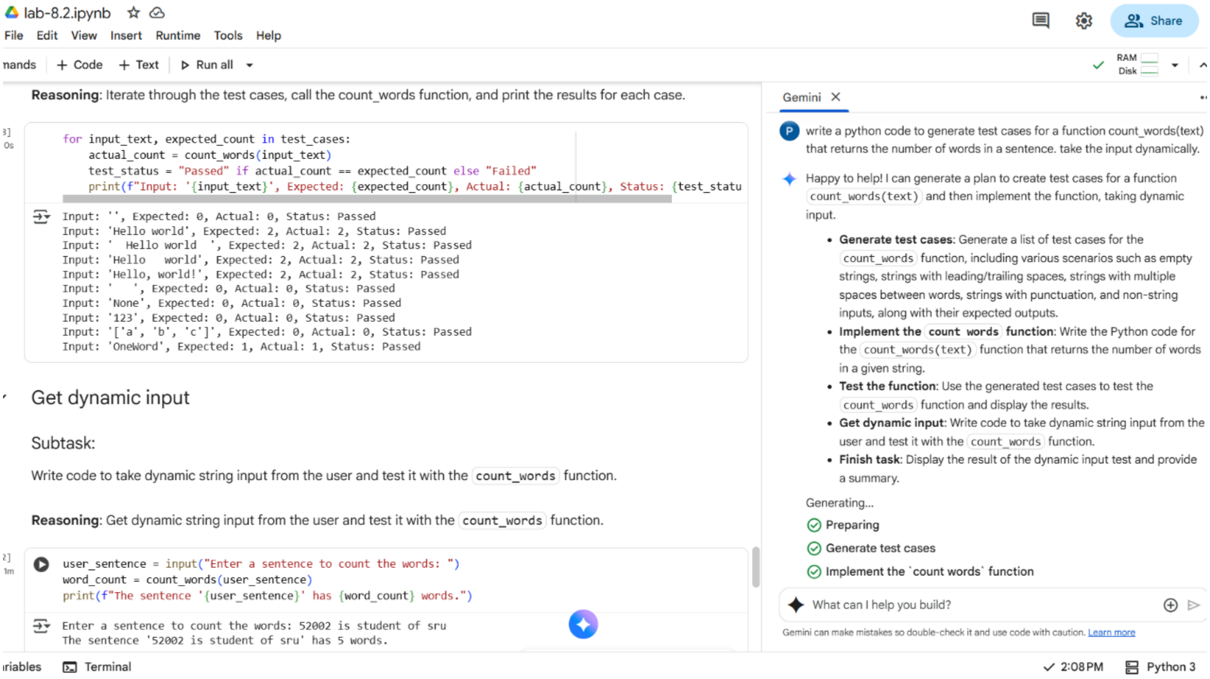
`count_words(text)` and then implement the function, taking dynamic input.

- **Generate test cases**: Generate a list of test cases for the `count_words` function, including various scenarios such as empty strings, strings with leading/trailing spaces, strings with multiple spaces between words, strings with punctuation, and non-string inputs, along with their expected outputs.
- **Implement the `count words` function**: Write the Python code for the `count_words(text)` function that returns the number of words in a given string.
- **Test the function**: Use the generated test cases to test the `count_words` function and display the results.
- **Get dynamic input**: Write code to take dynamic string input from the user and test it with the `count_words` function.
- **Finish task**: Display the result of the dynamic input test and provide a summary.

Generating...
✓ Preparing
✓ Generate test cases
✓ Implement the `count words` function
✓ Test the function
✓ Get dynamic input
✓ Finish task

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



Reasoning: Iterate through the test cases, call the count_words function, and print the results for each case.

```python
for input_text, expected_count in test_cases:
    actual_count = count_words(input_text)
    test_status = "Passed" if actual_count == expected_count else "Failed"
    print(f"Input: '{input_text}', Expected: {expected_count}, Actual: {actual_count}, Status: {test_statu
```

```
Input: '', Expected: 0, Actual: 0, Status: Passed
Input: 'Hello world', Expected: 2, Actual: 2, Status: Passed
Input: '  Hello world  ', Expected: 2, Actual: 2, Status: Passed
Input: 'Hello   world', Expected: 2, Actual: 2, Status: Passed
Input: 'Hello, world!', Expected: 2, Actual: 2, Status: Passed
Input: '   ', Expected: 0, Actual: 0, Status: Passed
Input: 'None', Expected: 0, Actual: 0, Status: Passed
Input: '123', Expected: 0, Actual: 0, Status: Passed
Input: '['a', 'b', 'c']', Expected: 0, Actual: 0, Status: Passed
Input: 'OneWord', Expected: 1, Actual: 1, Status: Passed
```

Get dynamic input

Subtask:

Write code to take dynamic string input from the user and test it with the `count_words` function.

Reasoning: Get dynamic string input from the user and test it with the `count_words` function.

```python
user_sentence = input("Enter a sentence to count the words: ")
word_count = count_words(user_sentence)
print(f"The sentence '{user_sentence}' has {word_count} words.")
```

```
Enter a sentence to count the words: 52002 is student of sru
The sentence '52002 is student of sru' has 5 words.
```

P write a python code to generate test cases for a function count_words(text) that returns the number of words in a sentence. take the input dynamically.

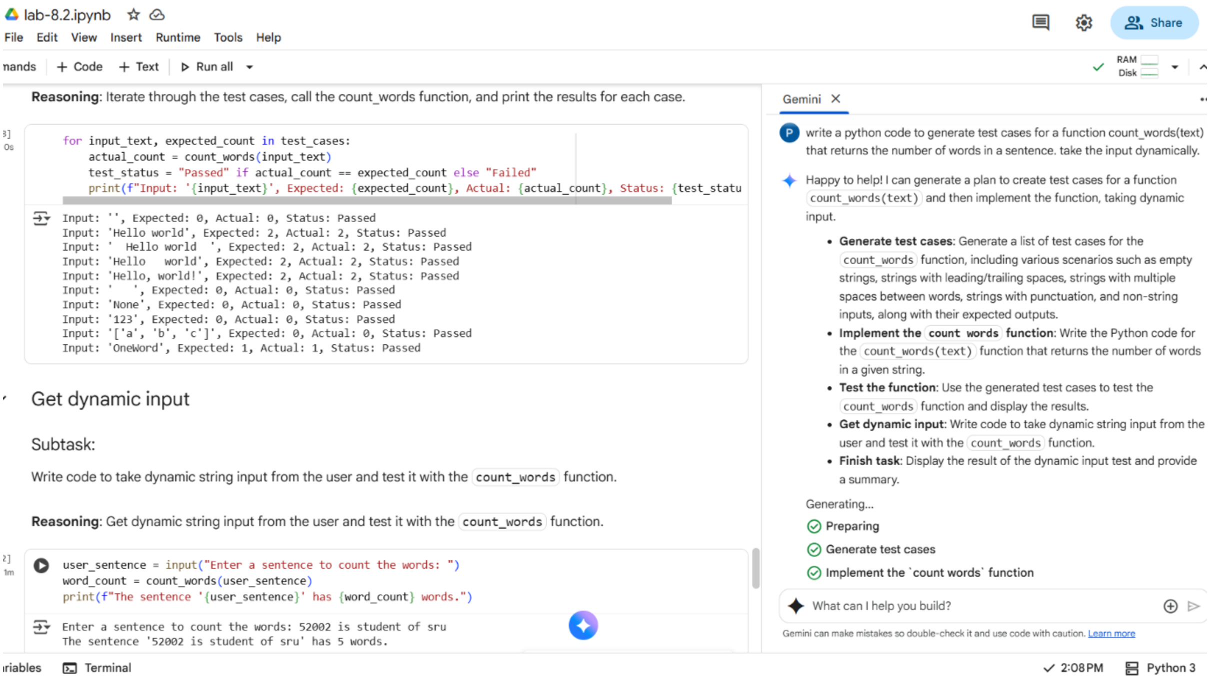
Happy to help! I can generate a plan to create test cases for a function `count_words(text)` and then implement the function, taking dynamic input.

- **Generate test cases**: Generate a list of test cases for the `count_words` function, including various scenarios such as empty strings, strings with leading/trailing spaces, strings with multiple spaces between words, strings with punctuation, and non-string inputs, along with their expected outputs.
- **Implement the `count words` function**: Write the Python code for the `count_words(text)` function that returns the number of words in a given string.
- **Test the function**: Use the generated test cases to test the `count_words` function and display the results.
- **Get dynamic input**: Write code to take dynamic string input from the user and test it with the `count_words` function.
- **Finish task**: Display the result of the dynamic input test and provide a summary.

Generating...
✓ Preparing
✓ Generate test cases
✓ Implement the `count words` function

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more


lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
nands + Code + Text Run all
Share
RAM
Disk
Gemini
iriables Terminal
2:08PM Python 3

Reasoning: Iterate through the test cases, call the count_words function, and print the results for each case.

```
for input_text, expected_count in test_cases:
    actual_count = count_words(input_text)
    test_status = "Passed" if actual_count == expected_count else "Failed"
    print(f"Input: '{input_text}', Expected: {expected_count}, Actual: {actual_count}, Status: {test_statu
```

```
Input: '', Expected: 0, Actual: 0, Status: Passed
Input: 'Hello world', Expected: 2, Actual: 2, Status: Passed
Input: '  Hello world  ', Expected: 2, Actual: 2, Status: Passed
Input: 'Hello   world', Expected: 2, Actual: 2, Status: Passed
Input: 'Hello, world!', Expected: 2, Actual: 2, Status: Passed
Input: '   ', Expected: 0, Actual: 0, Status: Passed
Input: 'None', Expected: 0, Actual: 0, Status: Passed
Input: '123', Expected: 0, Actual: 0, Status: Passed
Input: '['a', 'b', 'c']', Expected: 0, Actual: 0, Status: Passed
Input: 'OneWord', Expected: 1, Actual: 1, Status: Passed
```

Get dynamic input

Subtask:

Write code to take dynamic string input from the user and test it with the `count_words` function.

Reasoning: Get dynamic string input from the user and test it with the `count_words` function.

```
user_sentence = input("Enter a sentence to count the words: ")
word_count = count_words(user_sentence)
print(f"The sentence '{user_sentence}' has {word_count} words.")
```

```
Enter a sentence to count the words: 52002 is student of sru
The sentence '52002 is student of sru' has 5 words.
```

write a python code to generate test cases for a function count_words(text) that returns the number of words in a sentence. take the input dynamically.

Happy to help! I can generate a plan to create test cases for a function `count_words(text)` and then implement the function, taking dynamic input.

- **Generate test cases**: Generate a list of test cases for the `count_words` function, including various scenarios such as empty strings, strings with leading/trailing spaces, strings with multiple spaces between words, strings with punctuation, and non-string inputs, along with their expected outputs.
- **Implement the `count words` function**: Write the Python code for the `count_words(text)` function that returns the number of words in a given string.
- **Test the function**: Use the generated test cases to test the `count_words` function and display the results.
- **Get dynamic input**: Write code to take dynamic string input from the user and test it with the `count_words` function.
- **Finish task**: Display the result of the dynamic input test and provide a summary.

Generating...

✓ Preparing

✓ Generate test cases

✓ Implement the `count words` function

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Observation:

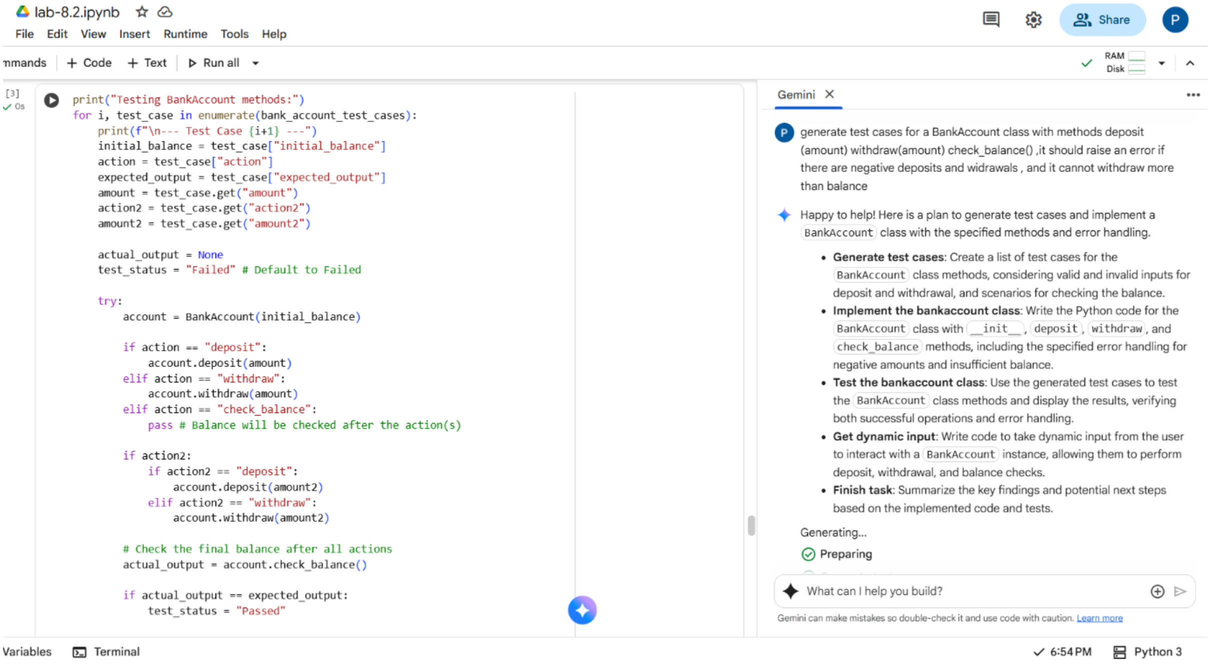
A comprehensive set of test cases was generated to evaluate the count_words function under various conditions, including edge cases like empty strings, strings with only spaces, and non-string inputs, as well as strings with punctuation and multiple spaces.

The dynamic input section correctly takes a sentence from the user and provides the word count using the implemented function.

Task-4:

Prompt: generate test cases for a BankAccount class with methods deposit (amount) withdraw(amount) check_balance() ,it should raise an error if there are negative deposits and widrawals , and it cannot withdraw more than balance

Code&Output:



lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
mmands + Code + Text ▷ Run all
Share
RAM
Disk
[3]
0s
Variables
Terminal
✓ 6:54 PM
Python 3

```
print("Testing BankAccount methods:")
for i, test_case in enumerate(bank_account_test_cases):
    print(f"\n--- Test Case {i+1} ---")
    initial_balance = test_case["initial_balance"]
    action = test_case["action"]
    expected_output = test_case["expected_output"]
    amount = test_case.get("amount")
    action2 = test_case.get("action2")
    amount2 = test_case.get("amount2")

    actual_output = None
    test_status = "Failed" # Default to Failed

    try:
        account = BankAccount(initial_balance)

        if action == "deposit":
            account.deposit(amount)
        elif action == "withdraw":
            account.withdraw(amount)
        elif action == "check_balance":
            pass # Balance will be checked after the action(s)

        if action2:
            if action2 == "deposit":
                account.deposit(amount2)
            elif action2 == "withdraw":
                account.withdraw(amount2)

        # Check the final balance after all actions
        actual_output = account.check_balance()

        if actual_output == expected_output:
            test_status = "Passed"
```

Gemini

generate test cases for a BankAccount class with methods deposit (amount) withdraw(amount) check_balance() ,it should raise an error if there are negative deposits and widrawals , and it cannot withdraw more than balance

Happy to help! Here is a plan to generate test cases and implement a `BankAccount` class with the specified methods and error handling.

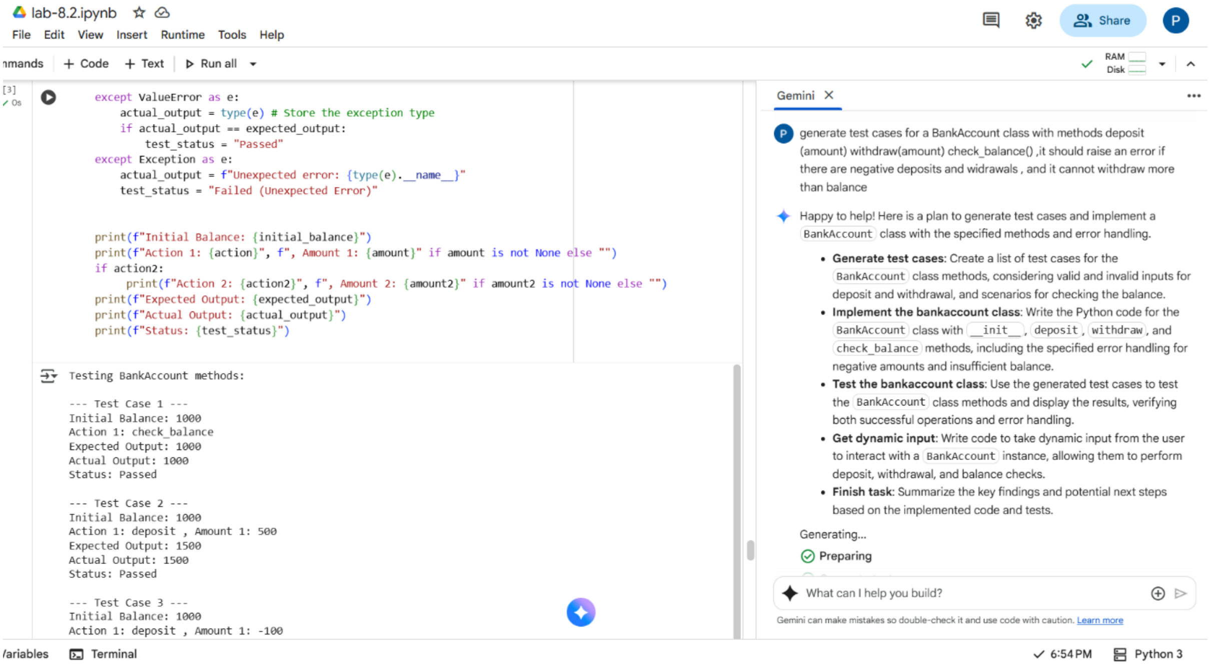
- **Generate test cases**: Create a list of test cases for the `BankAccount` class methods, considering valid and invalid inputs for deposit and withdrawal, and scenarios for checking the balance.
- **Implement the bankaccount class**: Write the Python code for the `BankAccount` class with `__init__`, `deposit`, `withdraw`, and `check_balance` methods, including the specified error handling for negative amounts and insufficient balance.
- **Test the bankaccount class**: Use the generated test cases to test the `BankAccount` class methods and display the results, verifying both successful operations and error handling.
- **Get dynamic input**: Write code to take dynamic input from the user to interact with a `BankAccount` instance, allowing them to perform deposit, withdrawal, and balance checks.
- **Finish task**: Summarize the key findings and potential next steps based on the implemented code and tests.

Generating...

✓ Preparing

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
mmands + Code + Text ▷ Run all
Share
RAM
Disk
[3]
0s
Variables
Terminal
✓ 6:54 PM
Python 3

```
    except ValueError as e:
        actual_output = type(e) # Store the exception type
        if actual_output == expected_output:
            test_status = "Passed"
    except Exception as e:
        actual_output = f"Unexpected error: {type(e).__name__}"
        test_status = "Failed (Unexpected Error)"


    print(f"Initial Balance: {initial_balance}")
    print(f"Action 1: {action}", f", Amount 1: {amount}" if amount is not None else "")
    if action2:
         print(f"Action 2: {action2}", f", Amount 2: {amount2}" if amount2 is not None else "")
    print(f"Expected Output: {expected_output}")
    print(f"Actual Output: {actual_output}")
    print(f"Status: {test_status}")
```

```
Testing BankAccount methods:

--- Test Case 1 ---
Initial Balance: 1000
Action 1: check_balance
Expected Output: 1000
Actual Output: 1000
Status: Passed

--- Test Case 2 ---
Initial Balance: 1000
Action 1: deposit , Amount 1: 500
Expected Output: 1500
Actual Output: 1500
Status: Passed

--- Test Case 3 ---
Initial Balance: 1000
Action 1: deposit , Amount 1: -100
```

Gemini

generate test cases for a BankAccount class with methods deposit (amount) withdraw(amount) check_balance() ,it should raise an error if there are negative deposits and widrawals , and it cannot withdraw more than balance

Happy to help! Here is a plan to generate test cases and implement a `BankAccount` class with the specified methods and error handling.

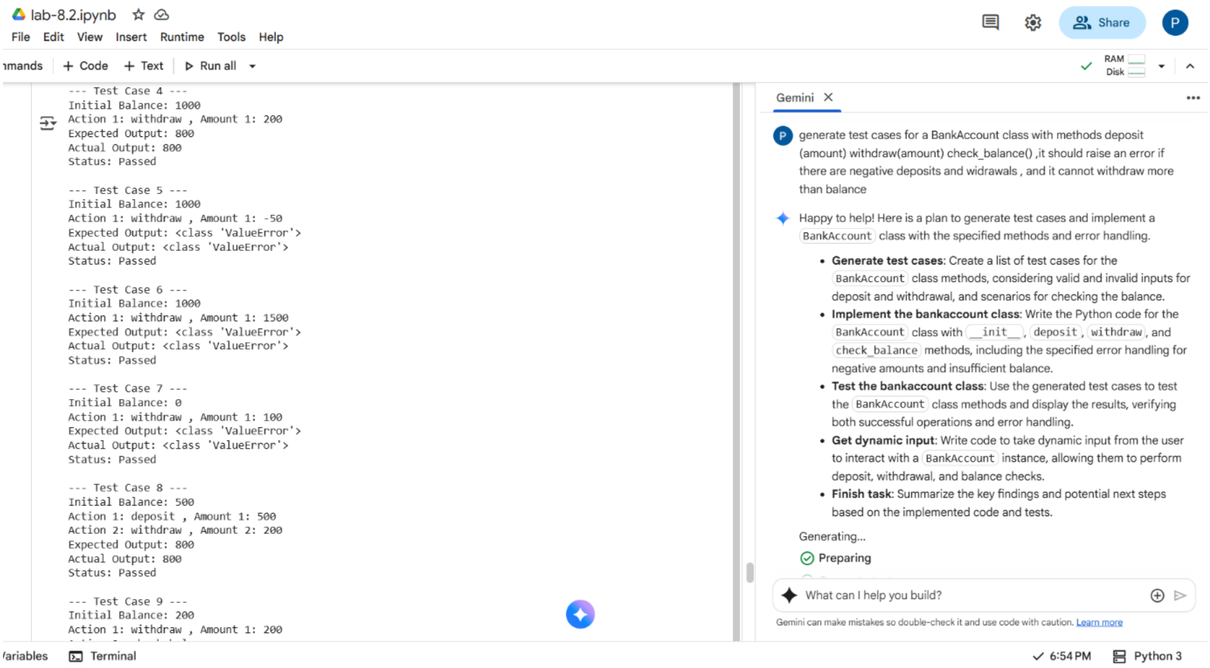
- **Generate test cases**: Create a list of test cases for the `BankAccount` class methods, considering valid and invalid inputs for deposit and withdrawal, and scenarios for checking the balance.
- **Implement the bankaccount class**: Write the Python code for the `BankAccount` class with `__init__`, `deposit`, `withdraw`, and `check_balance` methods, including the specified error handling for negative amounts and insufficient balance.
- **Test the bankaccount class**: Use the generated test cases to test the `BankAccount` class methods and display the results, verifying both successful operations and error handling.
- **Get dynamic input**: Write code to take dynamic input from the user to interact with a `BankAccount` instance, allowing them to perform deposit, withdrawal, and balance checks.
- **Finish task**: Summarize the key findings and potential next steps based on the implemented code and tests.

Generating...

✓ Preparing

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
--- Test Case 4 ---
Initial Balance: 1000
Action 1: withdraw , Amount 1: 200
Expected Output: 800
Actual Output: 800
Status: Passed

--- Test Case 5 ---
Initial Balance: 1000
Action 1: withdraw , Amount 1: -50
Expected Output: <class 'ValueError'>
Actual Output: <class 'ValueError'>
Status: Passed

--- Test Case 6 ---
Initial Balance: 1000
Action 1: withdraw , Amount 1: 1500
Expected Output: <class 'ValueError'>
Actual Output: <class 'ValueError'>
Status: Passed

--- Test Case 7 ---
Initial Balance: 0
Action 1: withdraw , Amount 1: 100
Expected Output: <class 'ValueError'>
Actual Output: <class 'ValueError'>
Status: Passed

--- Test Case 8 ---
Initial Balance: 500
Action 1: deposit , Amount 1: 500
Action 2: withdraw , Amount 2: 200
Expected Output: 800
Actual Output: 800
Status: Passed

--- Test Case 9 ---
Initial Balance: 200
Action 1: withdraw , Amount 1: 200
```

generate test cases for a BankAccount class with methods deposit (amount) withdraw(amount) check_balance() ,it should raise an error if there are negative deposits and widrawals , and it cannot withdraw more than balance

Happy to help! Here is a plan to generate test cases and implement a `BankAccount` class with the specified methods and error handling.

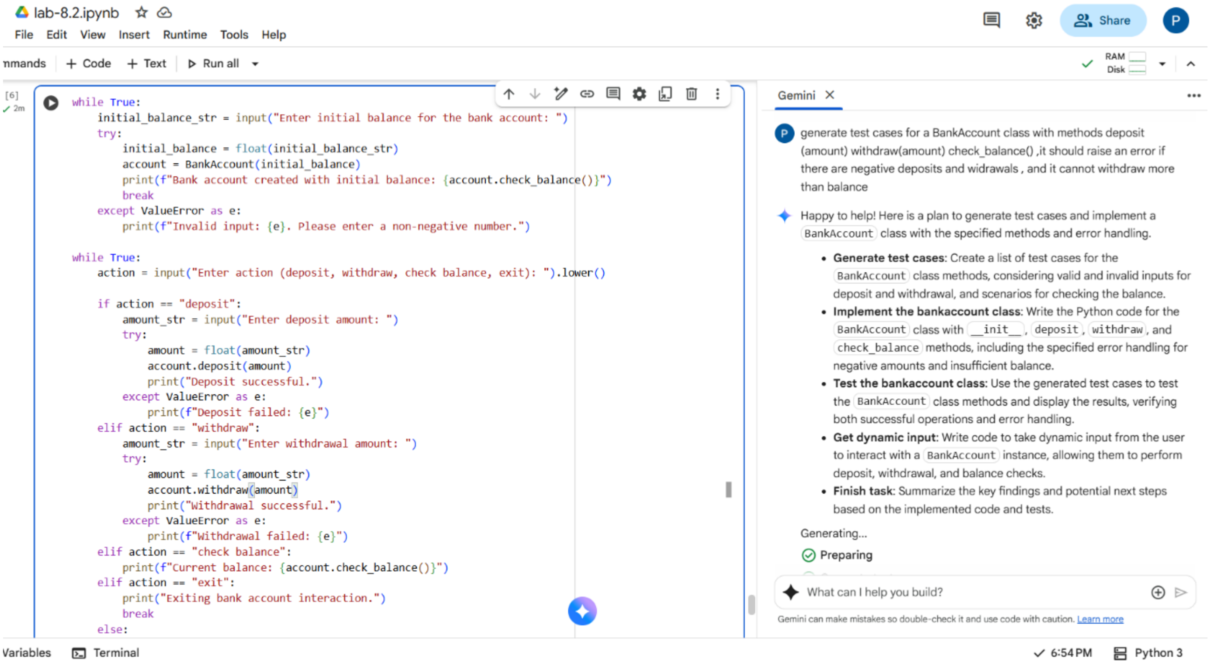
- **Generate test cases**: Create a list of test cases for the `BankAccount` class methods, considering valid and invalid inputs for deposit and withdrawal, and scenarios for checking the balance.
- **Implement the bankaccount class**: Write the Python code for the `BankAccount` class with `__init__`, `deposit`, `withdraw`, and `check_balance` methods, including the specified error handling for negative amounts and insufficient balance.
- **Test the bankaccount class**: Use the generated test cases to test the `BankAccount` class methods and display the results, verifying both successful operations and error handling.
- **Get dynamic input**: Write code to take dynamic input from the user to interact with a `BankAccount` instance, allowing them to perform deposit, withdrawal, and balance checks.
- **Finish task**: Summarize the key findings and potential next steps based on the implemented code and tests.

Generating...

✓ Preparing

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
while True:
    initial_balance_str = input("Enter initial balance for the bank account: ")
    try:
        initial_balance = float(initial_balance_str)
        account = BankAccount(initial_balance)
        print(f"Bank account created with initial balance: {account.check_balance()}")
        break
    except ValueError as e:
        print(f"Invalid input: {e}. Please enter a non-negative number.")

while True:
    action = input("Enter action (deposit, withdraw, check balance, exit): ").lower()

    if action == "deposit":
        amount_str = input("Enter deposit amount: ")
        try:
            amount = float(amount_str)
            account.deposit(amount)
            print("Deposit successful.")
        except ValueError as e:
            print(f"Deposit failed: {e}")
    elif action == "withdraw":
        amount_str = input("Enter withdrawal amount: ")
        try:
            amount = float(amount_str)
            account.withdraw(amount)
            print("Withdrawal successful.")
        except ValueError as e:
            print(f"Withdrawal failed: {e}")
    elif action == "check balance":
        print(f"Current balance: {account.check_balance()}")
    elif action == "exit":
        print("Exiting bank account interaction.")
        break
    else:
```

generate test cases for a BankAccount class with methods deposit (amount) withdraw(amount) check_balance() ,it should raise an error if there are negative deposits and widrawals , and it cannot withdraw more than balance

Happy to help! Here is a plan to generate test cases and implement a `BankAccount` class with the specified methods and error handling.

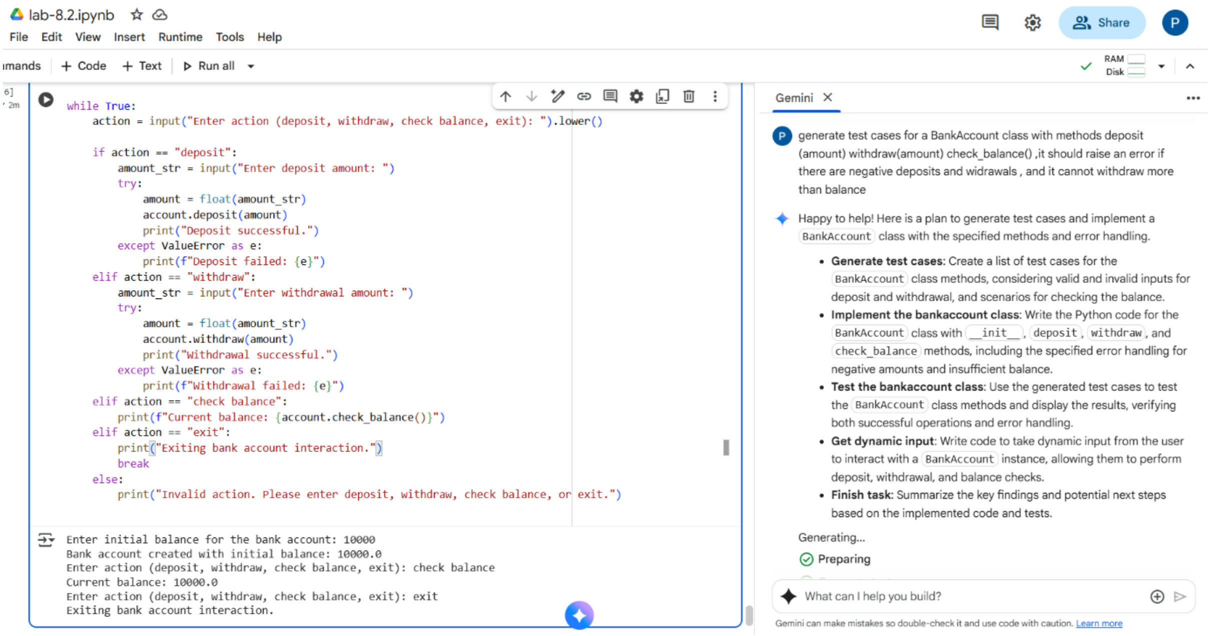
- **Generate test cases**: Create a list of test cases for the `BankAccount` class methods, considering valid and invalid inputs for deposit and withdrawal, and scenarios for checking the balance.
- **Implement the bankaccount class**: Write the Python code for the `BankAccount` class with `__init__`, `deposit`, `withdraw`, and `check_balance` methods, including the specified error handling for negative amounts and insufficient balance.
- **Test the bankaccount class**: Use the generated test cases to test the `BankAccount` class methods and display the results, verifying both successful operations and error handling.
- **Get dynamic input**: Write code to take dynamic input from the user to interact with a `BankAccount` instance, allowing them to perform deposit, withdrawal, and balance checks.
- **Finish task**: Summarize the key findings and potential next steps based on the implemented code and tests.

Generating...

✓ Preparing

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more


lab-8.2.ipynb
File Edit View Insert Runtime Tools Help
Share
ımands + Code + Text ▷ Run all
RAM
Disk
Gemini ✕

```
while True:
    action = input("Enter action (deposit, withdraw, check balance, exit): ").lower()

    if action == "deposit":
        amount_str = input("Enter deposit amount: ")
        try:
            amount = float(amount_str)
            account.deposit(amount)
            print("Deposit successful.")
        except ValueError as e:
            print(f"Deposit failed: {e}")
    elif action == "withdraw":
        amount_str = input("Enter withdrawal amount: ")
        try:
            amount = float(amount_str)
            account.withdraw(amount)
            print("Withdrawal successful.")
        except ValueError as e:
            print(f"Withdrawal failed: {e}")
    elif action == "check balance":
        print(f"Current balance: {account.check_balance()}")
    elif action == "exit":
        print("Exiting bank account interaction.")
        break
    else:
        print("Invalid action. Please enter deposit, withdraw, check balance, or exit.")
```

```
Enter initial balance for the bank account: 10000
Bank account created with initial balance: 10000.0
Enter action (deposit, withdraw, check balance, exit): check balance
Current balance: 10000.0
Enter action (deposit, withdraw, check balance, exit): exit
Exiting bank account interaction.
```

generate test cases for a BankAccount class with methods deposit (amount) withdraw(amount) check_balance() ,it should raise an error if there are negative deposits and widrawals , and it cannot withdraw more than balance

Happy to help! Here is a plan to generate test cases and implement a `BankAccount` class with the specified methods and error handling.

- **Generate test cases:** Create a list of test cases for the `BankAccount` class methods, considering valid and invalid inputs for deposit and withdrawal, and scenarios for checking the balance.
- **Implement the bankaccount class**: Write the Python code for the `BankAccount` class with `__init__`, `deposit`, `withdraw`, and `check_balance` methods, including the specified error handling for negative amounts and insufficient balance.
- **Test the bankaccount class**: Use the generated test cases to test the `BankAccount` class methods and display the results, verifying both successful operations and error handling.
- **Get dynamic input**: Write code to take dynamic input from the user to interact with a `BankAccount` instance, allowing them to perform deposit, withdrawal, and balance checks.
- **Finish task**: Summarize the key findings and potential next steps based on the implemented code and tests.

Generating...

Preparing

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Observation:

Comprehensive test cases covered various BankAccount scenarios, including error handling for invalid deposits, withdrawals, and insufficient funds.

The BankAccount class correctly implements deposit, withdraw, and check balance methods with appropriate error handling.

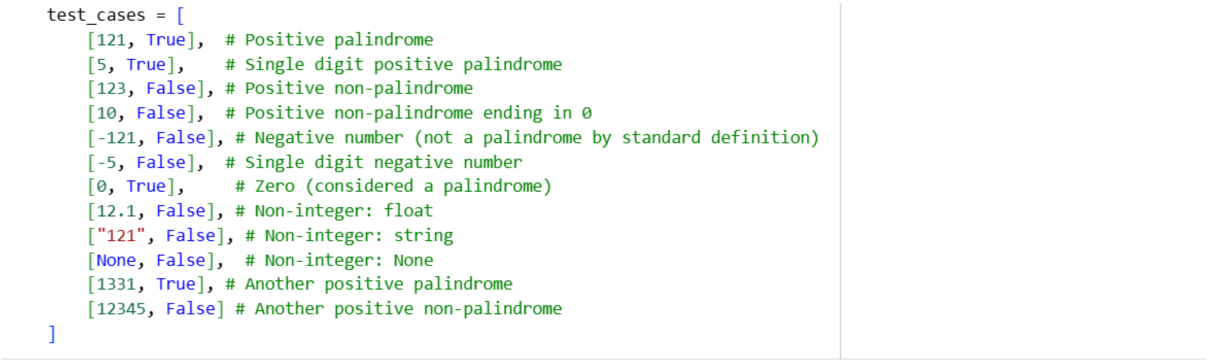
Testing confirmed the class handles valid operations and raises ValueError for invalid ones as expected.

Dynamic input allows interactive account management with clear feedback and error messages.

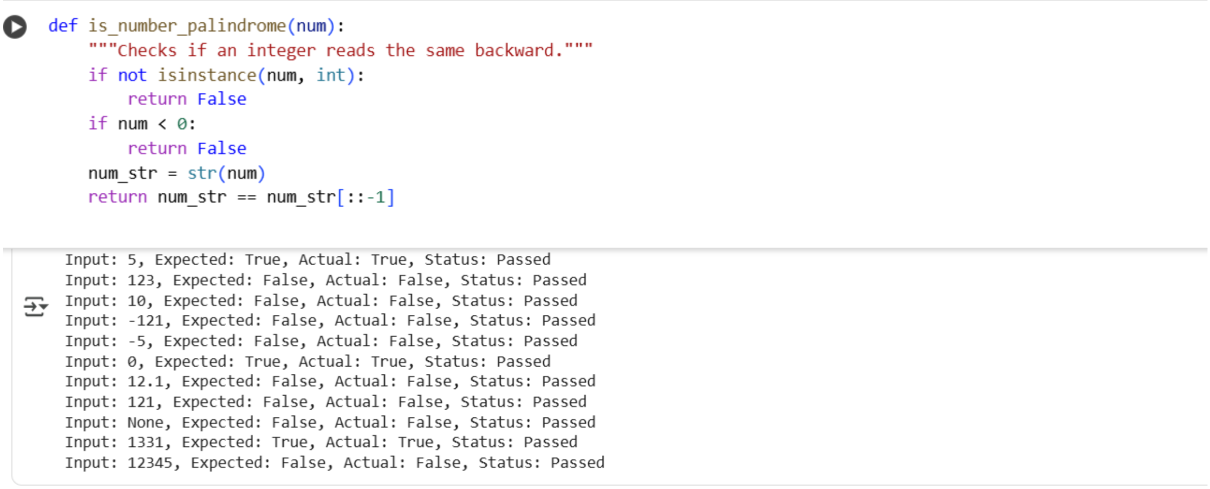
Task-5:

Prompt: write a python code to generate test cases for is_number_palindrome(num), which checks if an integer reads the same backward. take the input dynamically.

Code&Output:

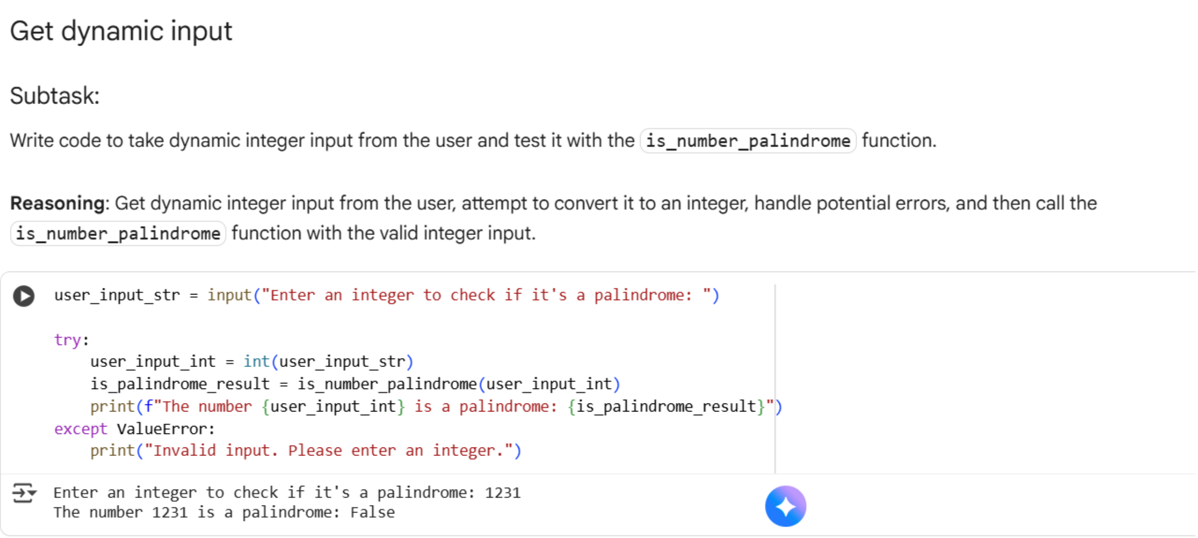


```
test_cases = [
    [121, True],  # Positive palindrome
    [5, True],    # Single digit positive palindrome
    [123, False], # Positive non-palindrome
    [10, False],  # Positive non-palindrome ending in 0
    [-121, False], # Negative number (not a palindrome by standard definition)
    [-5, False],  # Single digit negative number
    [0, True],     # Zero (considered a palindrome)
    [12.1, False], # Non-integer: float
    ["121", False], # Non-integer: string
    [None, False],  # Non-integer: None
    [1331, True], # Another positive palindrome
    [12345, False] # Another positive non-palindrome
]
```



```
def is_number_palindrome(num):
    """Checks if an integer reads the same backward."""
    if not isinstance(num, int):
        return False
    if num < 0:
        return False
    num_str = str(num)
    return num_str == num_str[::-1]
```

```
Input: 5, Expected: True, Actual: True, Status: Passed
Input: 123, Expected: False, Actual: False, Status: Passed
Input: 10, Expected: False, Actual: False, Status: Passed
Input: -121, Expected: False, Actual: False, Status: Passed
Input: -5, Expected: False, Actual: False, Status: Passed
Input: 0, Expected: True, Actual: True, Status: Passed
Input: 12.1, Expected: False, Actual: False, Status: Passed
Input: 121, Expected: False, Actual: False, Status: Passed
Input: None, Expected: False, Actual: False, Status: Passed
Input: 1331, Expected: True, Actual: True, Status: Passed
Input: 12345, Expected: False, Actual: False, Status: Passed
```



Get dynamic input

Subtask:

Write code to take dynamic integer input from the user and test it with the `is_number_palindrome` function.

Reasoning: Get dynamic integer input from the user, attempt to convert it to an integer, handle potential errors, and then call the `is_number_palindrome` function with the valid integer input.

```
user_input_str = input("Enter an integer to check if it's a palindrome: ")

try:
    user_input_int = int(user_input_str)
    is_palindrome_result = is_number_palindrome(user_input_int)
    print(f"The number {user_input_int} is a palindrome: {is_palindrome_result}")
except ValueError:
    print("Invalid input. Please enter an integer.")
```

```
Enter an integer to check if it's a palindrome: 1231
The number 1231 is a palindrome: False
```

Observation:

Comprehensive test cases covered various BankAccount scenarios, including error handling for invalid deposits, withdrawals, and insufficient funds

Dynamic input allows interactive account management with clear feedback and error messages.