
Program	:B.tech(CSE)
Specialization	:AIML
Course Title	:AI Assisted Coding
Course Code	:24CS002PC215
Semester	:3 rd semester
Academic Session	:2025-2026
Name of Student	:Kaveti Manohar
Enrollment No.	:2403A52079
Batch No.	:02
Date	:23/09/2025

LAB ASSIGNMENT

Task Description -1:

Task Description#1

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

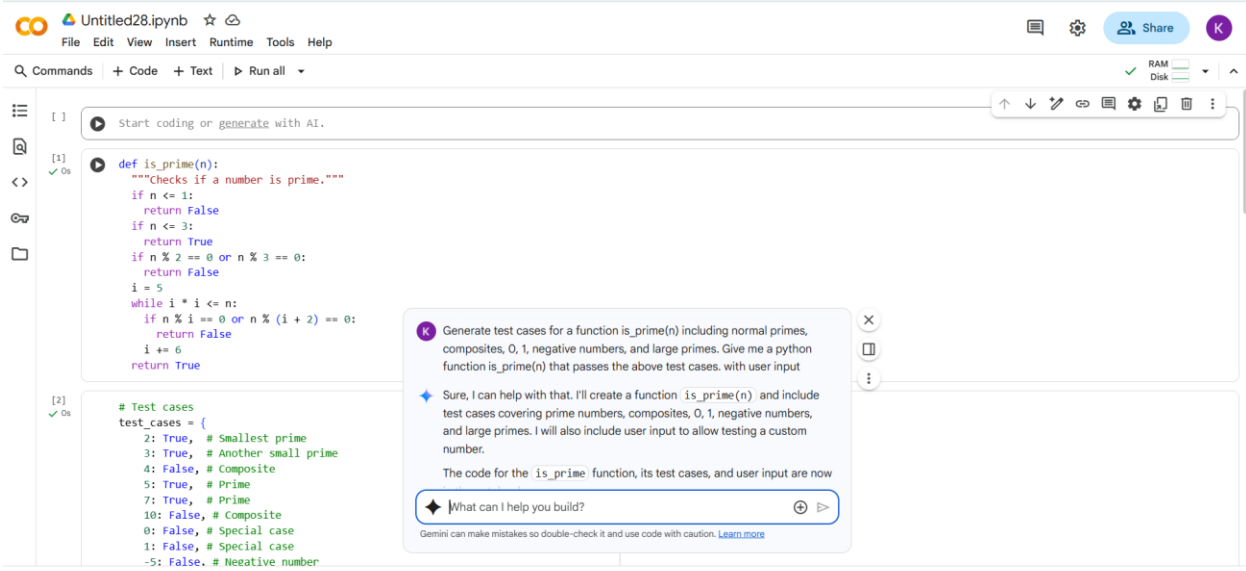
Requirements:

- Only integers > 1 can be prime.
- Check edge cases: 0, 1, 2, negative numbers, and large primes.

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:



The screenshot shows a Jupyter Notebook environment with the following content:

```
[1] ✓ Os
def is_prime(n):
    """Checks if a number is prime."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

[2] ✓ Os
# Test cases
test_cases = {
    2: True, # Smallest prime
    3: True, # Another small prime
    4: False, # Composite
    5: True, # Prime
    7: True, # Prime
    10: False, # Composite
    0: False, # Special case
    1: False, # Special case
    -5: False, # Negative number
}
```

An AI chat window is open, displaying the following prompt and response:

K 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

◆ Sure, I can help with that. I'll create a function `is_prime(n)` and include test cases covering prime numbers, composites, 0, 1, negative numbers, and large primes. I will also include user input to allow testing a custom number.

The code for the `is_prime` function, its test cases, and user input are now

◆ What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

➤ CODE With Output:

Untitled28.ipynb

File Edit View Insert Runtime Tools Help

Q Commands + Code + Text ▶ Run all

RAM Disk

Start coding or generate with AI.

[1] ✓ Os

```
def is_prime(n):
    """Checks if a number is prime."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

[2] ✓ Os

```
# Test cases
test_cases = [
    2: True, # Smallest prime
    3: True, # Another small prime
    4: False, # Composite
    5: True, # Prime
    7: True, # Prime
    10: False, # Composite
    0: False, # Special case
    1: False, # Special case
    -5: False, # Negative number
```

Variables Terminal

✓ 1:56 PM Python 3

Untitled28.ipynb

File Edit View Insert Runtime Tools Help

Q Commands + Code + Text ▶ Run all

RAM Disk

[2] ✓ Os

```
17: True, # Prime
29: True, # Prime
97: True, # Prime
101: True, # Prime
9973: True, # Large prime
10000: False, # Large composite
999983: True # A larger prime example (adjust for very large numbers as needed)

}

for number, expected in test_cases.items():
    result = is_prime(number)
    assert result == expected, f"is_prime({number}) failed: Expected {expected}, got {result}"
    print(f"is_prime({number}): {result} (Correct)")

is_prime(2): True (Correct)
is_prime(3): True (Correct)
is_prime(4): False (Correct)
is_prime(5): True (Correct)
is_prime(7): True (Correct)
is_prime(10): False (Correct)
is_prime(0): False (Correct)
is_prime(1): False (Correct)
is_prime(-5): False (Correct)
is_prime(17): True (Correct)
is_prime(29): True (Correct)
is_prime(97): True (Correct)
is_prime(101): True (Correct)
is_prime(9973): True (Correct)
is_prime(10000): False (Correct)
is_prime(999983): True (Correct)
```

Variables Terminal

✓ 1:56 PM Python 3

Untitled28.ipynb

File Edit View Insert Runtime Tools Help

Q Commands + Code + Text ▶ Run all

RAM Disk

```
is_prime(999983): True (Correct)
```

[3] ✓ 33s

```
# User input test
try:
    user_num_str = input("Enter a number to check if it's prime: ")
    user_num = int(user_num_str)
    if is_prime(user_num):
        print(f"{user_num} is a prime number.")
    else:
        print(f"{user_num} is not a prime number.")
except ValueError:
    print("Invalid input. Please enter an integer.")

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

Variables Terminal

✓ 1:56 PM Python 3

Summary:

The notebook contains Python code for checking if a number is prime.

1. **is_prime(n) function:** This function takes an integer `n` as input and returns `True` if `n` is a prime number, and `False` otherwise. It handles edge cases like numbers less than or equal to 1, and efficiently checks for divisibility by 2 and 3 before using a loop to check other potential divisors.
2. **Test Cases:** This section defines a dictionary `test_cases` with various numbers and their expected prime status (`True` or `False`). It then iterates through these test cases, calls the `is_prime` function for each number, and uses `assert` to verify that the result matches the expected outcome. This helps ensure the function works correctly for different scenarios.
3. **User Input Test:** This part allows the user to enter a number through an input prompt. It then calls the `is_prime` function with the user's number and prints

whether the entered number is prime or not. It also includes error handling to catch non-integer input.

Task Description -2:

Task Description#2 (Loops)

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

Requirements

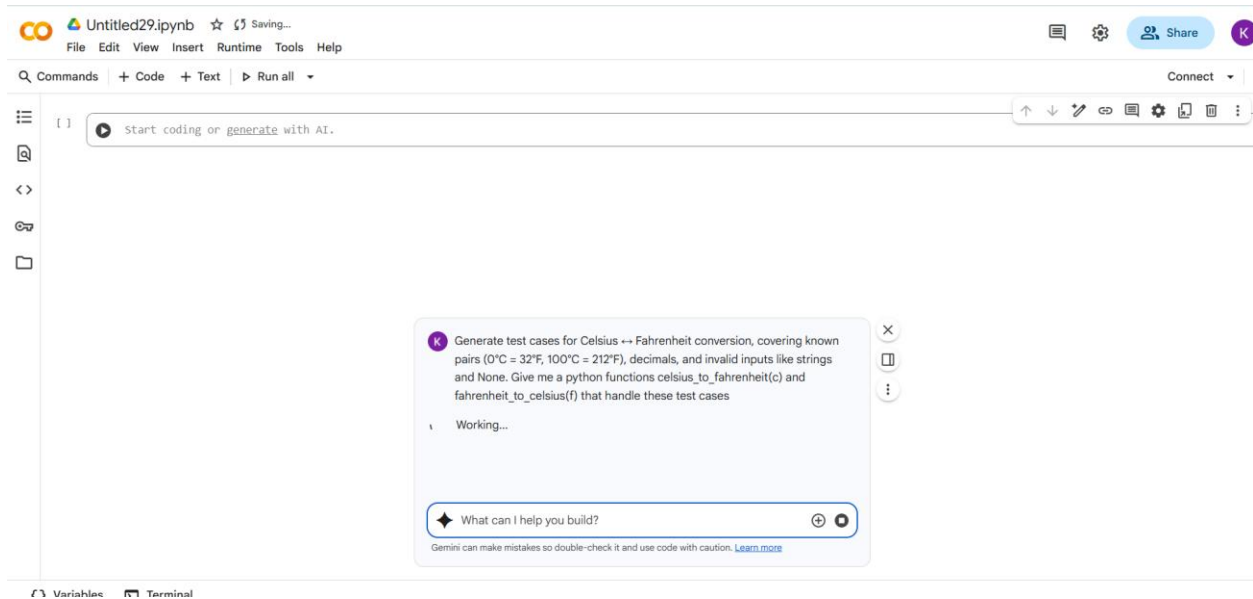
- Validate known pairs: $0^{\circ}\text{C} = 32^{\circ}\text{F}$, $100^{\circ}\text{C} = 212^{\circ}\text{F}$.
- Include decimals and invalid inputs like strings or None

PROMPT:

Generate test cases for Celsius \leftrightarrow Fahrenheit conversion, covering known pairs ($0^{\circ}\text{C} = 32^{\circ}\text{F}$, $100^{\circ}\text{C} = 212^{\circ}\text{F}$), 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.

QUESTION:



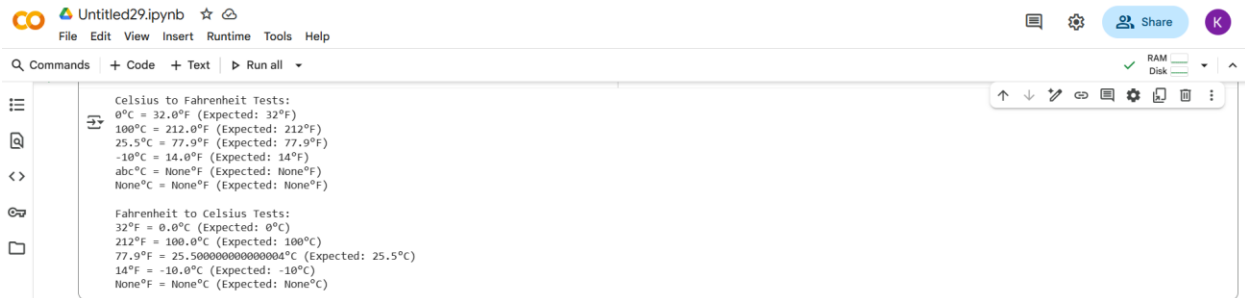
CODE:

```
[1] def celsius_to_fahrenheit(celsius):  
    """Converts Celsius to Fahrenheit.  
  
    Args:  
        celsius: The temperature in Celsius.  
  
    Returns:  
        The temperature in Fahrenheit, or None if the input is invalid.  
    """  
    if not isinstance(celsius, (int, float)):  
        return None  
    return (celsius * 9/5) + 32  
  
def fahrenheit_to_celsius(fahrenheit):  
    """Converts Fahrenheit to Celsius.  
  
    Args:  
        fahrenheit: The temperature in Fahrenheit.  
  
    Returns:  
        The temperature in Celsius, or None if the input is invalid.  
    """  
    if not isinstance(fahrenheit, (int, float)):  
        return None  
    return (fahrenheit - 32) * 5/9
```

Let's test these functions with various inputs.

```
[7] # Test cases  
test_cases = [  
    (0, 32), # Known pair: 0°C = 32°F  
    (100, 212), # Known pair: 100°C = 212°F  
    (25.5, 77.9), # Decimal  
    (-10, 14), # Negative temperature  
    ("abc", None), # Invalid input for Celsius to Fahrenheit  
    (None, None) # Invalid input for both  
]  
  
print("Celsius to Fahrenheit Tests:")  
for celsius, expected_fahrenheit in test_cases:  
    actual_fahrenheit = celsius_to_fahrenheit(celsius)  
    print(f"{celsius}°C = {actual_fahrenheit}°F (Expected: {expected_fahrenheit}°F)")  
    assert actual_fahrenheit == expected_fahrenheit, f"Error: {celsius}°C conversion failed."  
  
print("\nFahrenheit to Celsius Tests:")  
for expected_celsius, fahrenheit in test_cases:  
    # Skip the test case with invalid expected Celsius value  
    if expected_celsius == "abc":  
        continue  
    actual_celsius = fahrenheit_to_celsius(fahrenheit)  
    # Allow for small floating point differences, but handle None explicitly  
    if expected_celsius is None and actual_celsius is None:  
        assert True # Both are None, test passes  
    elif expected_celsius is not None and actual_celsius is not None:  
        assert abs(actual_celsius - expected_celsius) < 1e-9, f"Error: {fahrenheit}°F conversion failed."  
    else:  
        assert False, f"Error: {fahrenheit}°F conversion failed. Expected: {expected_celsius}°C, Got: {actual_celsius}°C"  
    print(f"{fahrenheit}°F = {actual_celsius}°C (Expected: {expected_celsius}°C)")
```

OUTPUT:



The screenshot shows a Jupyter Notebook titled 'Untitled29.ipynb'. The code cell contains two sections of tests. The first section, 'Celsius to Fahrenheit Tests', shows a list of tuples where each tuple contains a Celsius value, the actual Fahrenheit result, and the expected Fahrenheit value. The second section, 'Fahrenheit to Celsius Tests', shows a similar list of tuples. The interface includes a top menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with icons for commands, code, text, and running. On the left is a sidebar with icons for file explorer, search, and other notebook functions. On the right is a status bar showing 'RAM' and 'Disk' usage.

```
Celsius to Fahrenheit Tests:
0°C = 32.0°F (Expected: 32°F)
100°C = 212.0°F (Expected: 212°F)
25.5°C = 77.9°F (Expected: 77.9°F)
-10°C = 14.0°F (Expected: 14°F)
abc°C = None°F (Expected: None°F)
None°C = None°F (Expected: None°F)

Fahrenheit to Celsius Tests:
32°F = 0.0°C (Expected: 0°C)
212°F = 100.0°C (Expected: 100°C)
77.9°F = 25.500000000000004°C (Expected: 25.5°C)
14°F = -10.0°C (Expected: -10°C)
None°F = None°C (Expected: None°C)
```

EXPLANATION:

1. **test_cases list:** This list holds tuples, where each tuple represents a test case. The first element of each tuple is a Celsius value, and the second element is the corresponding Fahrenheit value. It includes various scenarios like known conversions, decimals, negative numbers, and invalid inputs (a string and None).
2. **Celsius to Fahrenheit Tests:**
 - The code iterates through the test_cases list.
 - For each tuple, it extracts the Celsius value and the expected Fahrenheit value.
 - It calls the celsius_to_fahrenheit() function with the Celsius value to get the actual Fahrenheit result.
 - It prints the input Celsius, the actual Fahrenheit output, and the expected Fahrenheit value.

- `assert actual_fahrenheit == expected_fahrenheit`: This line checks if the actual result matches the expected result. If they don't match, it raises an `AssertionError` with a message indicating which conversion failed.

3. Fahrenheit to Celsius Tests:

- This section also iterates through the `test_cases` list.
- It extracts the expected Celsius value and the Fahrenheit value from each tuple.
- `if expected_celsius == "abc": continue`: This line skips the test case where the expected Celsius value is the string "abc" because this test case is specifically designed for the Celsius to Fahrenheit function to test invalid input, and is not applicable for the Fahrenheit to Celsius conversion where the input is the Fahrenheit value.
- It calls the `fahrenheit_to_celsius()` function with the Fahrenheit value to get the actual Celsius result.
- It uses an `if/elif/else` block to handle assertions:
 - If both the expected and actual Celsius values are `None`, the assertion passes.

- If both are not None, it checks if the absolute difference between the actual and expected values is less than $1e-9$. This is done to account for potential small floating-point inaccuracies in calculations.
- If neither of the above conditions is met (meaning one is None and the other is not, or there's a significant difference in numerical values), the assertion fails.
- It prints the input Fahrenheit, the actual Celsius output, and the expected Celsius value

Task Description -3:

Task Description#3

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

Requirement

Handle normal text, multiple spaces, punctuation, and empty strings.

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:

The screenshot shows a Jupyter Notebook titled 'Untitled29.ipynb'. The code cell contains two sets of tests for a function `count_words(text)`.

Celsius to Fahrenheit Tests:

- `0°C = 32.0°F` (Expected: 32°F)
- `100°C = 212.0°F` (Expected: 212°F)
- `25.5°C = 77.9°F` (Expected: 77.9°F)
- `-10°C = 14.0°F` (Expected: 14°F)
- `abc°C = None°F` (Expected: None°F)
- `None°C = None°F` (Expected: None°F)

Fahrenheit to Celsius Tests:

- `32°F = 0.0°C` (Expected: 0°C)
- `212°F = 100.0°C` (Expected: 100°C)
- `77.9°F = 25.500000000000004°C` (Expected: 25.5°C)
- `14°F = -10.0°C` (Expected: -10°C)
- `None°F = None°C` (Expected: None°C)

A Gemini AI chat window is open in the foreground, displaying the following prompt and response:

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.

Response: Working...

Input field: What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

#CODE WITH OUTPUT:



The screenshot shows a Jupyter Notebook interface with a file named 'Untitled28.ipynb'. The active cell, labeled '[14]', contains a Python function definition for `count_words`. The function is designed to count the number of words in a given text string, handling multiple spaces and punctuation. The code includes a docstring, an `Args` section, a `Returns` section, and the function logic. The logic involves stripping leading and trailing whitespace, replacing punctuation with spaces, splitting the text into words, and filtering out any empty strings. The function is tested with a list of test cases in the cell below it.

```
[14]: import re

def count_words(text):
    """
    Counts the number of words in a sentence, handling multiple spaces and punctuation.

    Args:
        text: The input string.

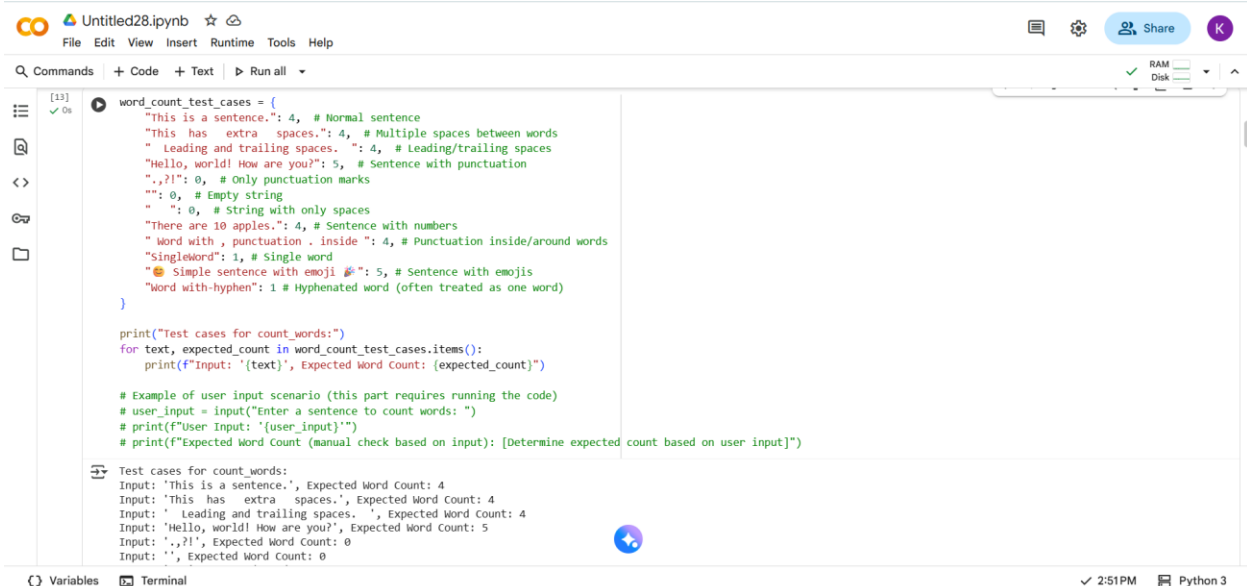
    Returns:
        The number of words in the string.
    """
    if not isinstance(text, str):
        return 0

    # Remove leading and trailing whitespace
    text = text.strip()

    # Replace all punctuation marks with spaces
    text = re.sub(r'[.,!?:;]', ' ', text)

    # Split the processed text into a list of words using whitespace as a delimiter
    words = text.split()

    # Filter out any empty strings that may result from the splitting process and return the count
    return len([word for word in words if word])
```



The screenshot shows a Jupyter Notebook interface with a file named 'Untitled28.ipynb'. The active cell, labeled '[13]', contains a dictionary of test cases for the `count_words` function. The dictionary, `word_count_test_cases`, lists various input strings and their expected word counts. The cell also includes code to print these test cases and a user input scenario for manual testing.

```
[13]: word_count_test_cases = {
    "This is a sentence.": 4, # Normal sentence
    "This has extra spaces.": 4, # Multiple spaces between words
    "Leading and trailing spaces. ": 4, # Leading/trailing spaces
    "Hello, world! How are you?": 5, # Sentence with punctuation
    ".?!": 0, # Only punctuation marks
    "": 0, # Empty string
    " ": 0, # String with only spaces
    "There are 10 apples.": 4, # Sentence with numbers
    "Word with , punctuation . inside ": 4, # Punctuation inside/around words
    "Singleword": 1, # Single word
    "Simple sentence with emoji 🐼": 5, # Sentence with emojis
    "Word with-hyphen": 1 # Hyphenated word (often treated as one word)
}

print("Test cases for count_words:")
for text, expected_count in word_count_test_cases.items():
    print(f'Input: '{text}', Expected Word Count: {expected_count}')

# Example of user input scenario (this part requires running the code)
# user_input = input("Enter a sentence to count words: ")
# print(f'User Input: '{user_input}')
# print(f'Expected Word Count (manual check based on input): {Determine expected count based on user input}')
```

EXPLANATION:

The code in the visible cells is related to testing a function called `count_words`.

1. **Cell 762959fa:** This cell defines a dictionary called `word_count_test_cases`. This dictionary holds

various example strings (sentences or phrases) as keys and the expected number of words in each string as the corresponding values. These test cases cover different scenarios like normal sentences, sentences with extra spaces, leading/trailing spaces, punctuation, only punctuation, empty strings, strings with only spaces, sentences with numbers, punctuation inside or around words, single words, sentences with emojis, and hyphenated words. The code then iterates through this dictionary and prints each input string along with its expected word count. This helps in verifying the correctness of the `count_words` function when it's implemented and tested.

Task Description -4:

Task Description#4

- Generate test cases for a `BankAccount` class with:

Methods:

`deposit(amount)`

`withdraw(amount)`

`check_balance()`

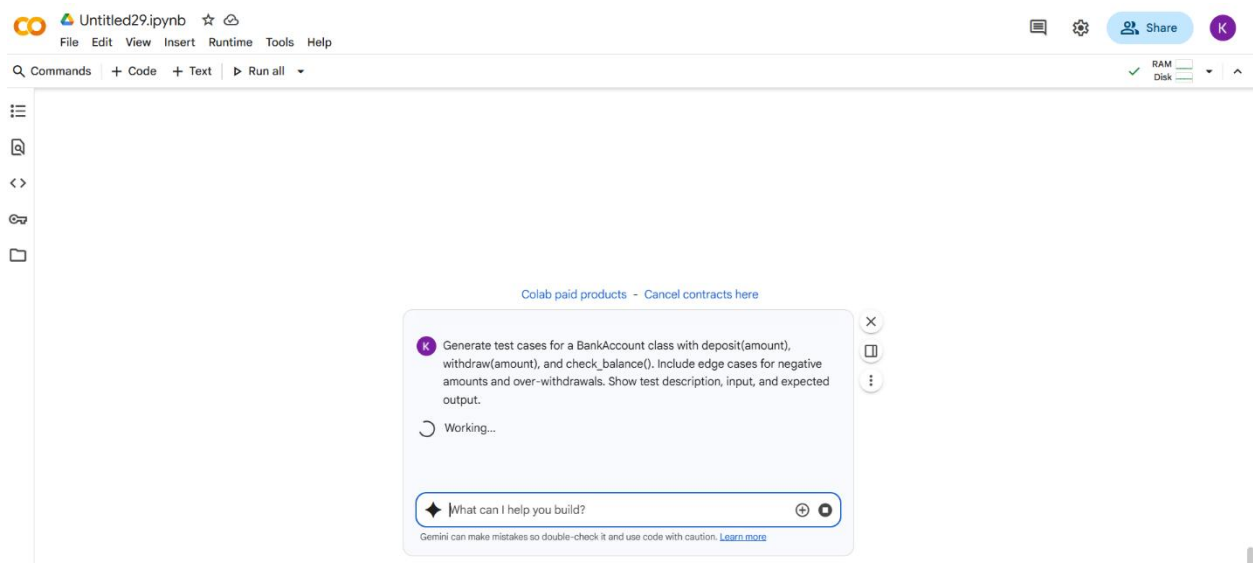
Requirements:

- Negative deposits/withdrawals should raise an error.
- Cannot withdraw more than balance.

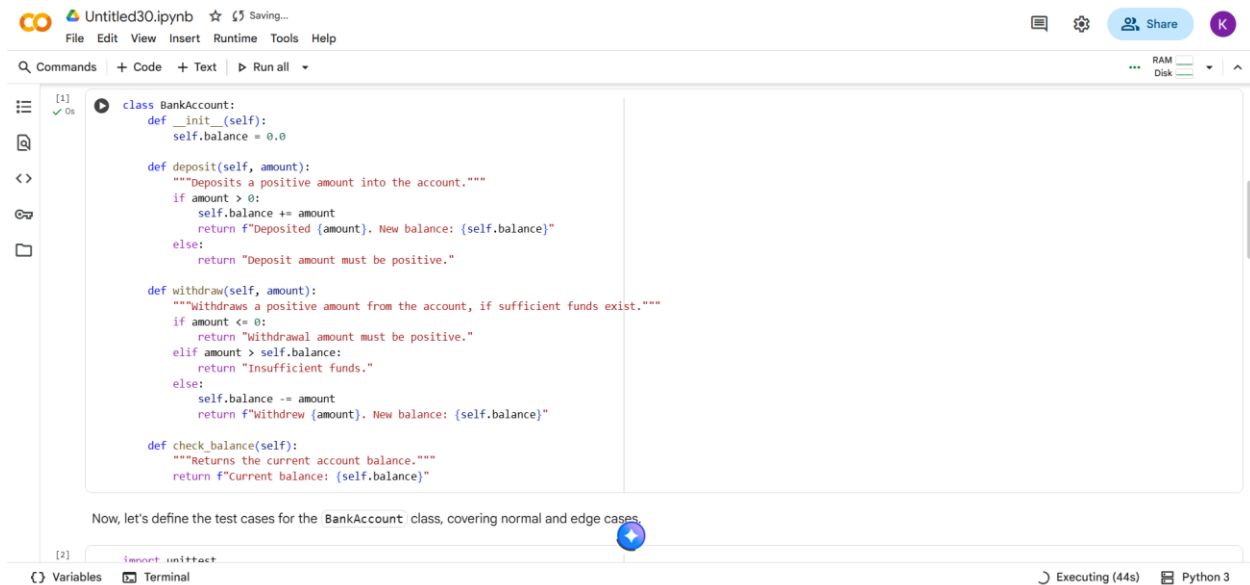
PROMPT:

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:



Untitled30.ipynb ☆ Saving...

File Edit View Insert Runtime Tools Help

Q Commands + Code + Text ▶ Run all

[1] ✓ 0s

```
class BankAccount:
    def __init__(self):
        self.balance = 0.0

    def deposit(self, amount):
        """Deposits a positive amount into the account."""
        if amount > 0:
            self.balance += amount
            return f"Deposited {amount}. New balance: {self.balance}"
        else:
            return "Deposit amount must be positive."

    def withdraw(self, amount):
        """Withdraws a positive amount from the account, if sufficient funds exist."""
        if amount <= 0:
            return "Withdrawal amount must be positive."
        elif amount > self.balance:
            return "Insufficient funds."
        else:
            self.balance -= amount
            return f"Withdrew {amount}. New balance: {self.balance}"

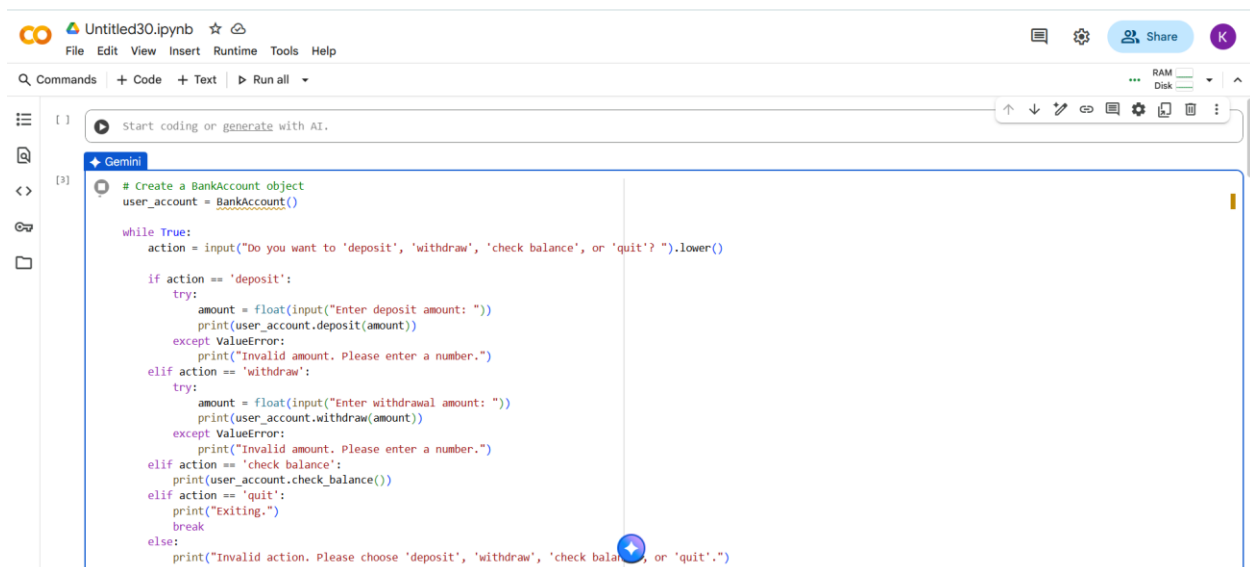
    def check_balance(self):
        """Returns the current account balance."""
        return f"Current balance: {self.balance}"
```

Now, let's define the test cases for the `BankAccount` class, covering normal and edge cases.

[2]

Variables Terminal

Executing (44s) Python 3



Untitled30.ipynb ☆

File Edit View Insert Runtime Tools Help

Q Commands + Code + Text ▶ Run all

[1] Start coding or generate with AI.

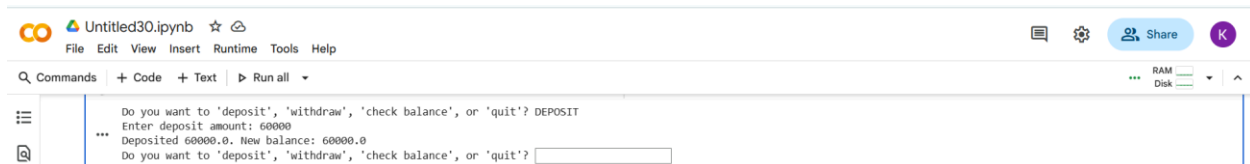
[2] Gemini

```
# Create a BankAccount object
user_account = BankAccount()

while True:
    action = input("Do you want to 'deposit', 'withdraw', 'check balance', or 'quit'? ").lower()

    if action == 'deposit':
        try:
            amount = float(input("Enter deposit amount: "))
            print(user_account.deposit(amount))
        except ValueError:
            print("Invalid amount. Please enter a number.")
    elif action == 'withdraw':
        try:
            amount = float(input("Enter withdrawal amount: "))
            print(user_account.withdraw(amount))
        except ValueError:
            print("Invalid amount. Please enter a number.")
    elif action == 'check balance':
        print(user_account.check_balance())
    elif action == 'quit':
        print("Exiting.")
        break
    else:
        print("Invalid action. Please choose 'deposit', 'withdraw', 'check balance', or 'quit'.")
```

OUTPUT:



Untitled30.ipynb ☆

File Edit View Insert Runtime Tools Help

Q Commands + Code + Text ▶ Run all

Do you want to 'deposit', 'withdraw', 'check balance', or 'quit'? DEPOSIT
Enter deposit amount: 60000
Deposited 60000.0. New balance: 60000.0
Do you want to 'deposit', 'withdraw', 'check balance', or 'quit'?

EXPLANATION:

This code cell allows you to interact with a BankAccount object through the console. Here's a breakdown:

1. **user_account = BankAccount():** This line creates a new instance of the BankAccount class, essentially creating a new bank account with an initial balance of 0.0.
2. **while True::** This starts an infinite loop, allowing you to perform multiple actions until you choose to quit.
3. **action = input(...):** This line prompts you to enter an action (deposit, withdraw, check balance, or quit). The `.lower()` converts your input to lowercase for easier comparison.
4. **if action == 'deposit':** If you enter 'deposit', it prompts you for an amount, converts it to a float, and calls the `deposit()` method of your `user_account`. It includes a try-except block to handle cases where you might enter non-numeric input.
5. **elif action == 'withdraw':** If you enter 'withdraw', it works similarly to the deposit section, prompting for an amount and calling the `withdraw()` method. It also includes error handling for non-numeric input.
6. **elif action == 'check balance':** If you enter 'check balance', it calls the `check_balance()` method and prints the current balance.

7. **elif action == 'quit':** If you enter 'quit', it prints an exit message and the break statement exits the while loop, ending the program.
8. **else::** If you enter anything other than the valid actions, it prints an "Invalid action" message.

Task Description -5:

Task Description#5

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

Examples:

121 → True

123 → False

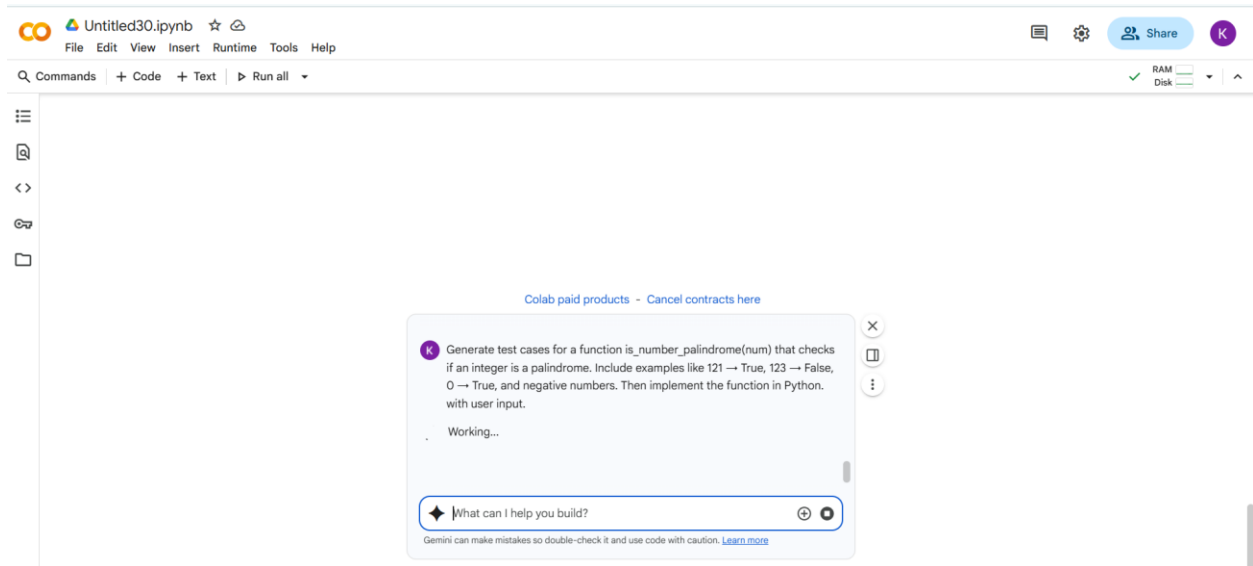
0, negative numbers → handled gracefully

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:

Untitled30.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

RAM Disk

Implement the function

Subtask:

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

Reasoning: Implement the `is_number_palindrome` function as described in the instructions.

[5] ✓ 0s

```
def is_number_palindrome(num):  
    """  
    Checks if an integer is a palindrome.  
  
    Args:  
        num: The integer to check.  
  
    Returns:  
        True if the number is a palindrome, False otherwise.  
    """  
    if num < 0:  
        return False  
    num_str = str(num)  
    return num_str == num_str[::-1]
```

Add user input

Variables Terminal

4:46 PM Python 3

Untitled30.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

RAM Disk

[4] ✓ 0s

```
test_cases = [  
    {"input": 121, "expected_output": True, "description": "Positive palindrome"},  
    {"input": 123, "expected_output": False, "description": "Positive non-palindrome"},  
    {"input": 0, "expected_output": True, "description": "Zero"},  
    {"input": 7, "expected_output": True, "description": "Single-digit number"},  
    {"input": -121, "expected_output": False, "description": "Negative number"},  
    {"input": -101, "expected_output": False, "description": "Negative palindrome (should be False)"},  
    {"input": 12321, "expected_output": True, "description": "Larger positive palindrome"},  
    {"input": 12345, "expected_output": False, "description": "Larger positive non-palindrome"},  
]
```

Implement the function

Untitled30.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

RAM Disk

✓ 0s

```
return num_str == num_str[::-1]
```

Add user input

Subtask:

Modify the code to take an integer as input from the user and then call the `is_number_palindrome()` function with the user's input.

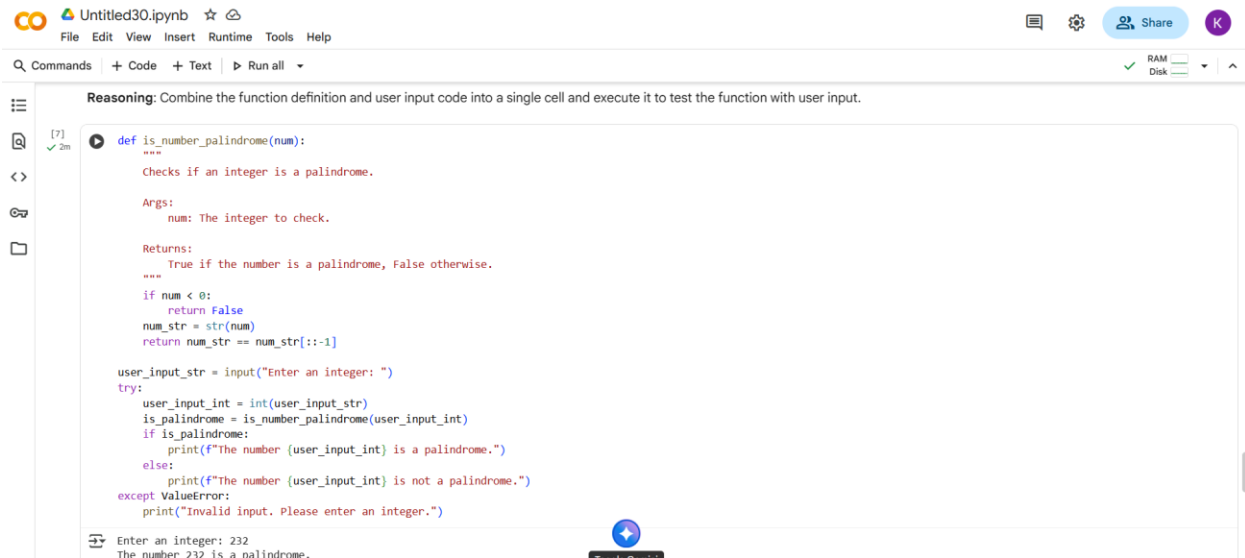
Reasoning: The subtask requires getting user input, converting it to an integer, calling the `is_number_palindrome` function, and printing the result. This can be done in a single code block.

[6] ✓ 20s

```
user_input_str = input("Enter an integer: ")  
try:  
    user_input_int = int(user_input_str)  
    is_palindrome = is_number_palindrome(user_input_int)  
    if is_palindrome:  
        print(f"The number {user_input_int} is a palindrome.")  
    else:  
        print(f"The number {user_input_int} is not a palindrome.")  
except ValueError:  
    print("Invalid input. Please enter an integer.")
```

Enter an integer: 232
The number 232 is a palindrome.

Combine and test



The screenshot shows a Jupyter Notebook titled 'Untitled30.ipynb'. The interface includes a top bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help' menus. Below the menu bar is a toolbar with icons for 'Commands', 'Code', 'Text', and 'Run all'. On the right side of the toolbar, there are icons for 'Share' and a user profile 'K'. The main area of the notebook contains a code cell with the following Python code:

```
[7] ✓ 2m def is_number_palindrome(num):  
    """  
    Checks if an integer is a palindrome.  
  
    Args:  
        num: The integer to check.  
  
    Returns:  
        True if the number is a palindrome, False otherwise.  
    """  
    if num < 0:  
        return False  
    num_str = str(num)  
    return num_str == num_str[::-1]  
  
    user_input_str = input("Enter an integer: ")  
    try:  
        user_input_int = int(user_input_str)  
        is_palindrome = is_number_palindrome(user_input_int)  
        if is_palindrome:  
            print(f"The number {user_input_int} is a palindrome.")  
        else:  
            print(f"The number {user_input_int} is not a palindrome.")  
    except ValueError:  
        print("Invalid input. Please enter an integer.")
```

Below the code cell, the output is displayed: "Enter an integer: 232" followed by "The number 232 is a palindrome." on the next line. A small blue circular icon with a white arrow is visible at the bottom right of the code cell.

EXPLANATION:

This code defines a function `is_number_palindrome` that checks if an integer is a palindrome. It also includes code to get input from the user and use the function.

Here's a breakdown:

1. **`def is_number_palindrome(num):`**: This defines the function that takes an integer `num` as input.

2. **if num < 0:** This checks if the number is negative. Negative numbers are not considered palindromes in this implementation, so it returns False.
3. **num_str = str(num):** The number is converted to a string so that it can be easily reversed and compared.
4. **return num_str == num_str[::-1]:** This is the core of the palindrome check. It compares the string representation of the number with its reverse (`[::-1]` creates a reversed copy of the string). If they are the same, the number is a palindrome and the function returns True; otherwise, it returns False.
5. **user_input_str = input("Enter an integer: "):**
This line prompts the user to enter an integer and stores their input as a string.
6. **try...except ValueError:** This block attempts to convert the user's input string to an integer. If the input is not a valid integer, a `ValueError` is caught, and an error message is printed.

7. **user_input_int = int(user_input_str):** If the input is valid, it's converted to an integer.

8. **is_palindrome =**

is_number_palindrome(user_input_int):

The `is_number_palindrome` function is called with the user's integer input, and the result (True or False) is stored in the `is_palindrome` variable.

9. **if is_palindrome::** This checks the value of `is_palindrome` and prints a message indicating whether the entered number is a palindrome or not.

-----THANK YOU-----