

AI ASSISTED CODING

LAB ASSIGNMENT – 8.2

SAHASRA VEERAGONI

2403A53007

24BTCAICYB01-2-1

#TASK-1

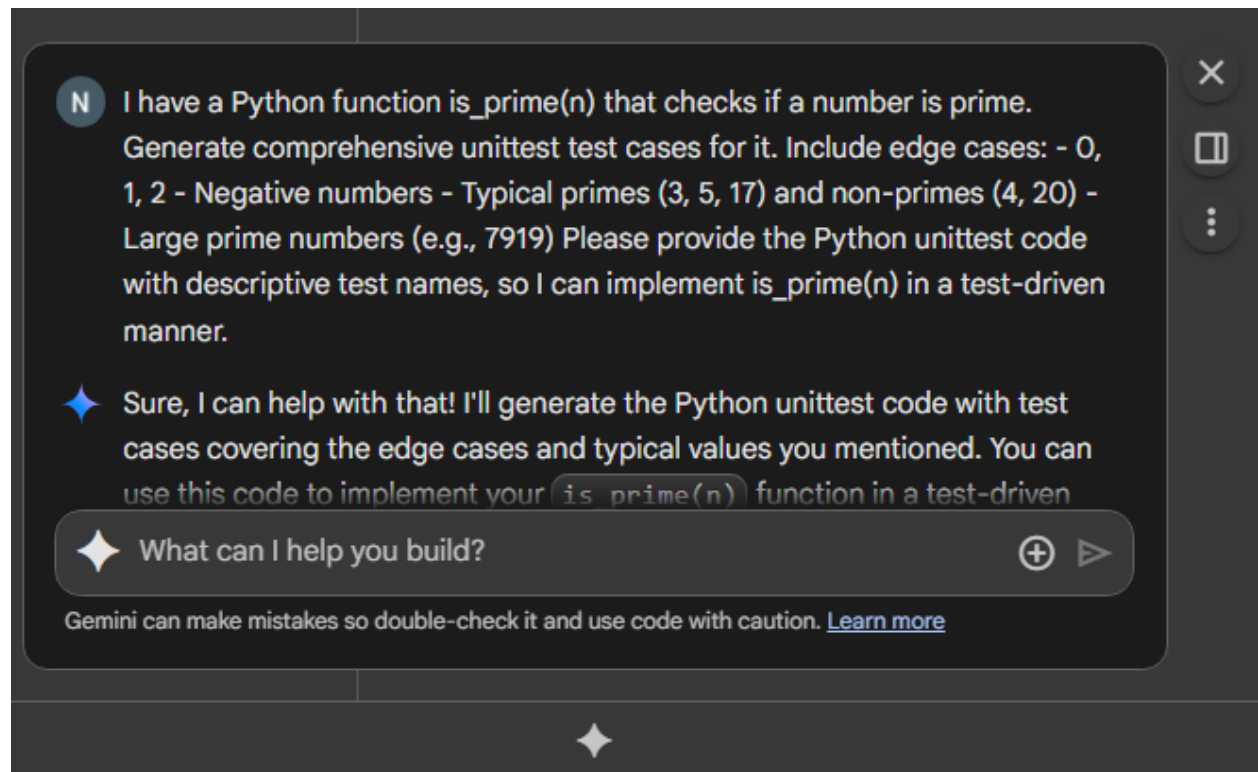
Use AI to generate test cases for a function `is_prime(n)` and then implement the function.

Requirements:

- Only integers > 1 can be prime.
-

Check edge cases: 0, 1, 2, negative numbers, and large primes..

PROMPT:



OUTPUT:

```
def is_prime(n):
    """
    Checks if a number is prime.

    Args:
        n: An integer.

    Returns:
        True if n is prime, False otherwise.
    """
    if not isinstance(n, int) or 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 cases
print(f"is_prime(0): {is_prime(0)}")
print(f"is_prime(1): {is_prime(1)}")
print(f"is_prime(2): {is_prime(2)}")
print(f"is_prime(-5): {is_prime(-5)}")
print(f"is_prime(7): {is_prime(7)}")
print(f"is_prime(10): {is_prime(10)}")
print(f"is_prime(97): {is_prime(97)}")
print(f"is_prime(100): {is_prime(100)}")

is_prime(0): False
is_prime(1): False
is_prime(2): True
is_prime(-5): False
is_prime(7): True
is_prime(10): False
is_prime(97): True
is_prime(100): False
```

Start coding or generate with AI.

Here are some test cases for the `is_prime(n)` function:

- `n = 0`: Should return `False` (not an integer > 1)
- `n = 1`: Should return `False` (not an integer > 1)
- `n = 2`: Should return `True` (smallest prime number)
- `n = -5`: Should return `False` (negative number)
- `n = 7`: Should return `True` (a small prime number)
- `n = 10`: Should return `False` (a composite number)
- `n = 97`: Should return `True` (a larger prime number)
- `n = 100`: Should return `False` (a larger composite number)

Here are some test cases for the `celsius_to_fahrenheit(c)` function:

- `c = 0`: Should return `32.0`
- `c = 100`: Should return `212.0`
- `c = 25.5`: Should return `77.9` (decimal input)
- `c = -10`: Should return `14.0` (negative input)
- `c = "abc"`: Should handle invalid input (e.g., return `None` or raise an error)
- `c = None`: Should handle invalid input

#TASK-2

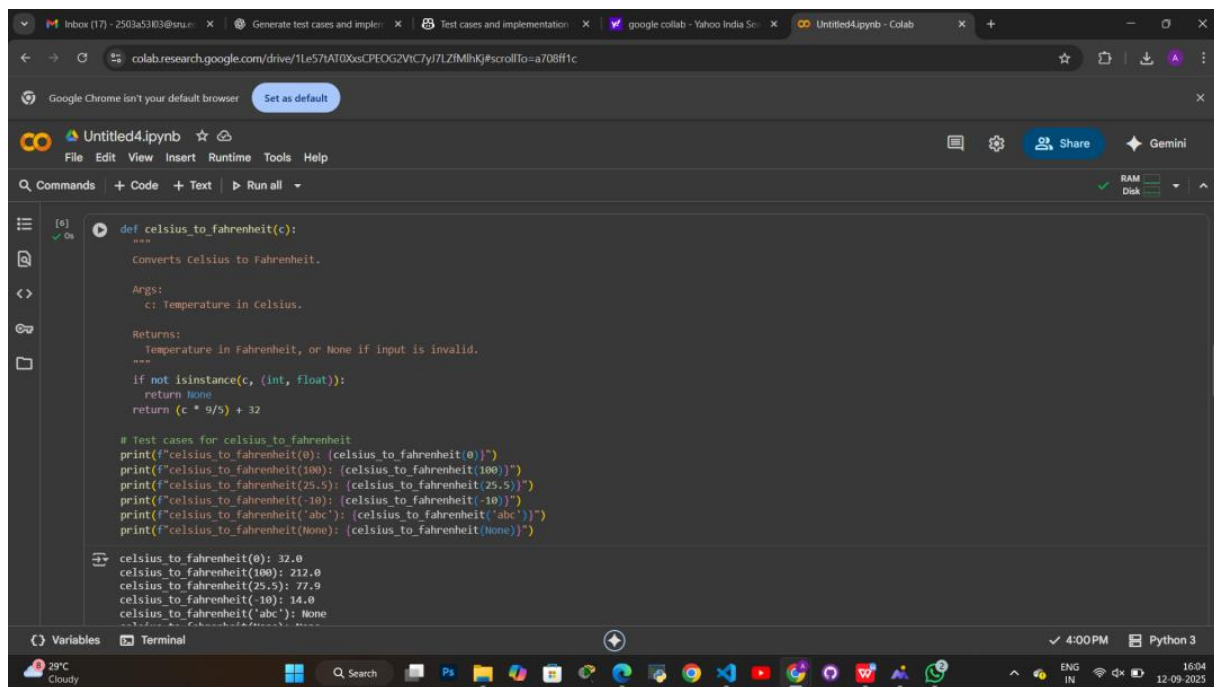
Task Description#2 (Loops)

- 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

OUTPUT:



```
[0] def celsius_to_fahrenheit(c):  
    """  
    Converts Celsius to Fahrenheit.  
  
    Args:  
        c: Temperature in Celsius.  
  
    Returns:  
        Temperature in Fahrenheit, or None if input is invalid.  
    """  
    if not isinstance(c, (int, float)):  
        return None  
    return (c * 9/5) + 32  
  
# Test cases for celsius_to_fahrenheit  
print(f"celsius_to_fahrenheit(0): {celsius_to_fahrenheit(0)}")  
print(f"celsius_to_fahrenheit(100): {celsius_to_fahrenheit(100)}")  
print(f"celsius_to_fahrenheit(25.5): {celsius_to_fahrenheit(25.5)}")  
print(f"celsius_to_fahrenheit(-10): {celsius_to_fahrenheit(-10)}")  
print(f"celsius_to_fahrenheit('abc'): {celsius_to_fahrenheit('abc')}")  
print(f"celsius_to_fahrenheit(None): {celsius_to_fahrenheit(None)}")  
  
celsius_to_fahrenheit(0): 32.0  
celsius_to_fahrenheit(100): 212.0  
celsius_to_fahrenheit(25.5): 77.9  
celsius_to_fahrenheit(-10): 14.0  
celsius_to_fahrenheit('abc'): None  
celsius_to_fahrenheit(None): None
```

Here are some test cases for the `fahrenheit_to_celsius(f)` function:

- `f = 32`: Should return `0.0`
- `f = 212`: Should return `100.0`
- `f = 77.9`: Should return `25.5` (decimal input)
- `f = 14`: Should return `-10.0` (negative input)
- `f = "xyz"`: Should handle invalid input (e.g., return `None` or raise an error)
- `f = None`: Should handle invalid input

#TASK-3

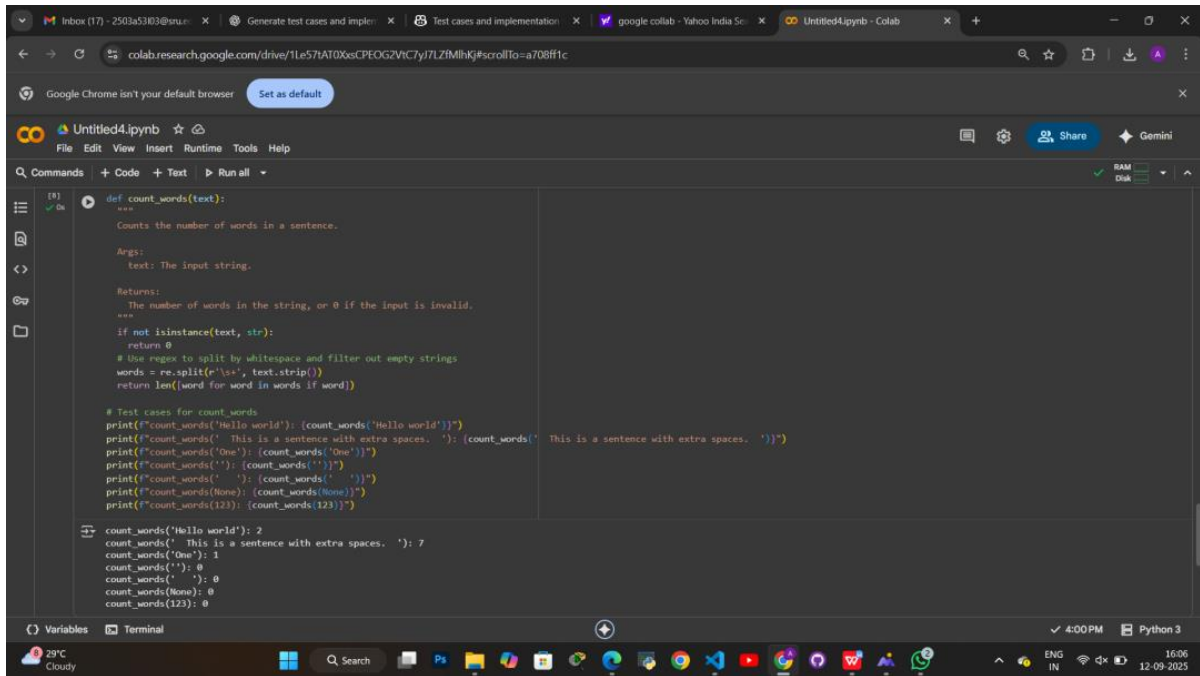
Task Description#3

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

OUTPUT:



The screenshot shows a Google Colab notebook titled 'Untitled4.ipynb'. The code defines a function `count_words(text)` that counts the number of words in a string. The function includes docstrings for its purpose, arguments, and return value. It uses `re.split` to split the string by whitespace and filters out empty strings. Below the function definition, there are several test cases using `print` statements to verify the function's behavior for various inputs, including 'Hello world', a sentence with extra spaces, 'One', an empty string, a string with only spaces, `None`, and an integer `123`. The output of the notebook shows the results of these tests, confirming that the function returns the correct word counts for each case.

```
[8] def count_words(text):  
    """  
    Counts the number of words in a sentence.  
  
    Args:  
        text: The input string.  
  
    Returns:  
        The number of words in the string, or 0 if the input is invalid.  
    """  
    if not isinstance(text, str):  
        return 0  
    # Use regex to split by whitespace and filter out empty strings  
    words = re.split(r'\s+', text.strip())  
    return len([word for word in words if word])  
  
    # Test cases for count_words  
    print(f"count_words('Hello world'): {count_words('Hello world')}")  
    print(f"count_words(' This is a sentence with extra spaces. '): {count_words(' This is a sentence with extra spaces. ')}")  
    print(f"count_words('One'): {count_words('One')}")  
    print(f"count_words(''): {count_words('')}")  
    print(f"count_words(' '): {count_words(' ')}")  
    print(f"count_words(None): {count_words(None)}")  
    print(f"count_words(123): {count_words(123)}")  
  
count_words('Hello world'): 2  
count_words(' This is a sentence with extra spaces. '): 7  
count_words('One'): 1  
count_words(''): 0  
count_words(' '): 0  
count_words(None): 0  
count_words(123): 0
```

Here are some test cases for the `count_words(text)` function:

- `text = "Hello world"`: Should return `2`
- `text = " This is a sentence with extra spaces. "`: Should return `7` (handling leading/trailing and multiple internal spaces)
- `text = "One"`: Should return `1`
- `text = ""`: Should return `0` (empty string)
- `text = " "`: Should return `0` (only spaces)
- `text = None`: Should handle invalid input (e.g., return `0` or raise an error)
- `text = 123`: Should handle invalid input

#TASK-4

Generate test cases for a BankAccount class with:

Methods:

`deposit(amount)`

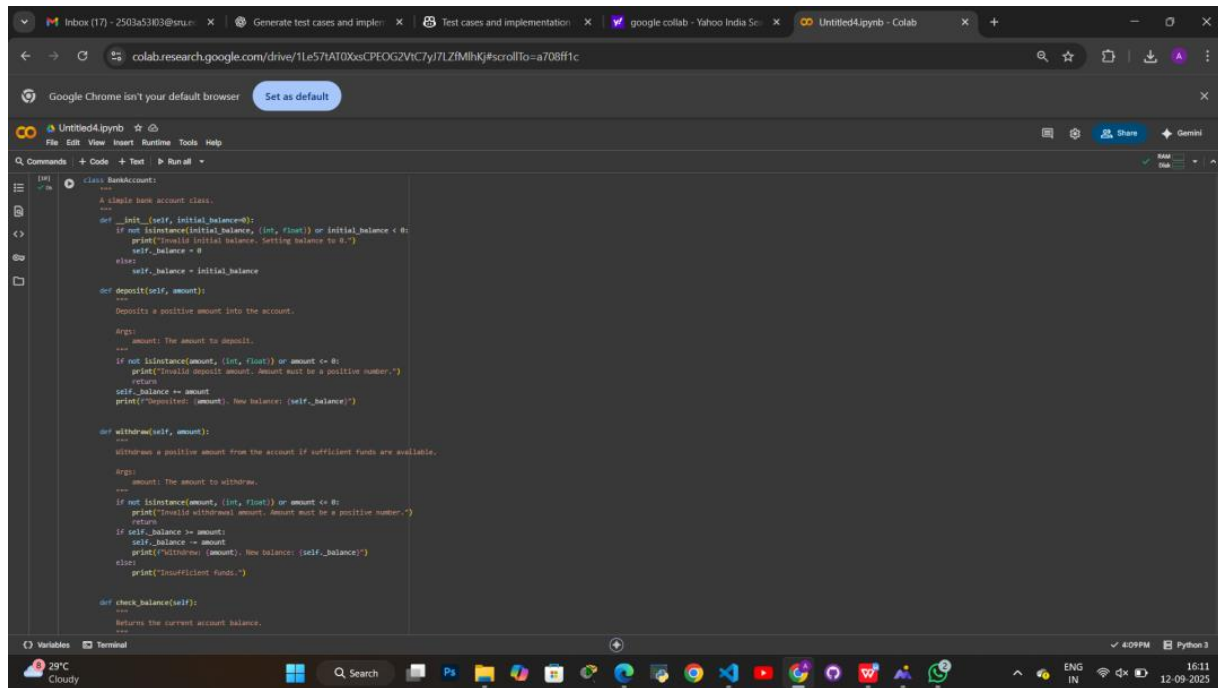
`withdraw(amount)`

`check_balance()`

Requirements:

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

OUTPUT:



```
class BankAccount:
    """
    A simple bank account class.
    """
    def __init__(self, initial_balance=0):
        """
        Initialize the account with an initial balance.
        """
        if not isinstance(initial_balance, (int, float)) or initial_balance < 0:
            print("Invalid initial balance. Setting balance to 0.")
            self._balance = 0
        else:
            self._balance = initial_balance

    def deposit(self, amount):
        """
        Deposit a positive amount into the account.

        Args:
            amount: The amount to deposit.

        """
        if not isinstance(amount, (int, float)) or amount <= 0:
            print("Invalid deposit amount. Amount must be a positive number.")
            return
        self._balance += amount
        print(f"Deposited: {amount}. New balance: {self._balance}")

    def withdraw(self, amount):
        """
        Withdraw a positive amount from the account if sufficient funds are available.

        Args:
            amount: The amount to withdraw.

        """
        if not isinstance(amount, (int, float)) or amount <= 0:
            print("Invalid withdrawal amount. Amount must be a positive number.")
            return
        if self._balance >= amount:
            self._balance -= amount
            print(f"Withdrew: {amount}. New balance: {self._balance}")
        else:
            print("Insufficient funds.")

    def check_balance(self):
        """
        Returns the current account balance.
        """
```

```
return self._balance

# Test cases for BankAccount
# Initial balance
account = BankAccount(100)
print(f"Initial balance: {account.check_balance()}")

# Deposit
account.deposit(50)
print(f"Balance after deposit: {account.check_balance()}")

# Multiple deposits
account.deposit(25.5)
account.deposit(75)
print(f"Balance after multiple deposits: {account.check_balance()}")

# Withdrawal within balance
account.withdrawal(50)
print(f"Balance after withdrawal: {account.check_balance()}")

# Multiple withdrawals
account.withdrawal(10.5)
account.withdrawal(5)
print(f"Balance after multiple withdrawals: {account.check_balance()}")

# Withdrawal exceeding balance
account.withdrawal(200)
print(f"Balance after exceeding withdrawal attempt: {account.check_balance()}")

# Zero deposit
account.deposit(0)
print(f"Balance after zero deposit: {account.check_balance()}")

# Zero withdrawal
account.withdrawal(0)
print(f"Balance after zero withdrawal: {account.check_balance()}")

# Negative deposit
account.deposit(-50)
print(f"Balance after negative deposit attempt: {account.check_balance()}")

# Negative withdrawal
account.withdrawal(-50)
print(f"Balance after negative withdrawal attempt: {account.check_balance()}")
```

```
# Negative withdrawal
account.withdrawal(-20)
print(f"Balance after negative withdrawal attempt: {account.check_balance()}")

# Non-numeric deposit
account.deposit("abc")
print(f"Balance after non-numeric deposit attempt: {account.check_balance()}")

# Non-numeric withdrawal
account.withdrawal("xyz")
print(f"Balance after non-numeric withdrawal attempt: {account.check_balance()}")

# Invalid initial balance
invalid_account = BankAccount(50)
print(f"Initial balance of invalid account: {invalid_account.check_balance()}")
```

Initial balance: 100
Deposit: 50, New balance: 150
Balance after deposit: 150
Deposit: 25.5, New balance: 175.5
Deposit: 75, New balance: 250.5
Balance after multiple deposits: 250.5
Withdrawal: 50, New balance: 200.5
Balance after withdrawal: 200.5
Withdrawal: 10.5, New balance: 189.5
Withdrawal: 5, New balance: 184.5
Balance after multiple withdrawals: 184.5
Balance after exceeding withdrawal attempt: 5.0
Invalid deposit amount. Amount must be a positive number.
Balance after zero deposit: 5.0
Invalid withdrawal amount. Amount must be a positive number.
Balance after zero withdrawal: 5.0
Invalid deposit amount. Amount must be a positive number.
Balance after negative deposit attempt: 5.0
Invalid withdrawal amount. Amount must be a positive number.
Balance after negative withdrawal attempt: 5.0
Invalid deposit amount. Amount must be a positive number.
Balance after non-numeric deposit attempt: 5.0
Invalid withdrawal amount. Amount must be a positive number.
Balance after non-numeric withdrawal attempt: 5.0
Invalid initial balance. Setting balance to 0.
Initial balance of invalid account: 0

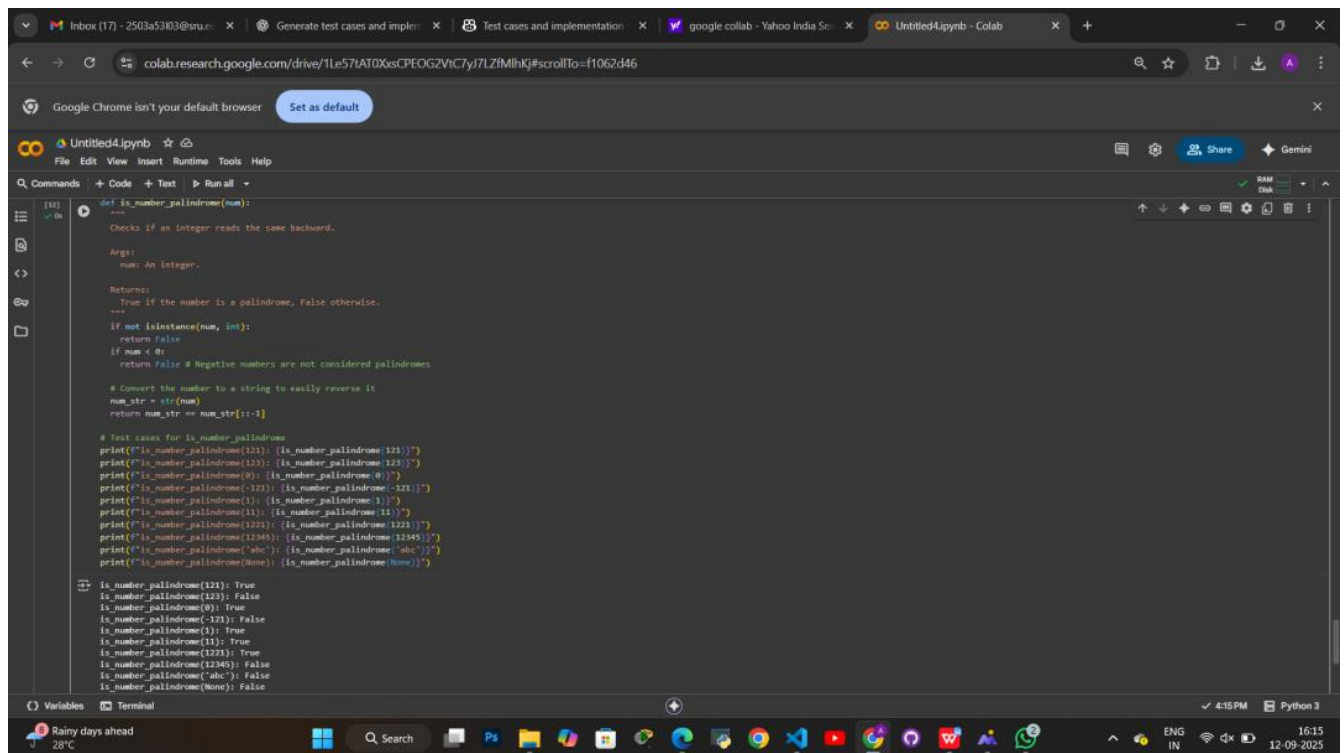
Here are some test cases for the `BankAccount` class:

- **Initial balance:** Create an account with an initial balance (e.g., 100). `check_balance()` should return the initial balance.
- **Deposit:** Deposit a positive amount (e.g., 50). `check_balance()` should reflect the increased balance (150).
- **Multiple deposits:** Make several deposits. `check_balance()` should show the cumulative balance.
- **Withdrawal within balance:** Withdraw an amount less than or equal to the current balance (e.g., 30). `check_balance()` should reflect the decreased balance (120).
- **Multiple withdrawals:** Make several withdrawals within the balance. `check_balance()` should be updated accordingly.
- **Withdrawal exceeding balance:** Attempt to withdraw an amount greater than the current balance (e.g., 200). The withdrawal should be denied, and the balance should remain unchanged.
- **Zero deposit:** Attempt to deposit zero. The balance should remain unchanged.
- **Zero withdrawal:** Attempt to withdraw zero. The balance should remain unchanged.
- **Negative deposit:** Attempt to deposit a negative amount. The deposit should be denied, and the balance should remain unchanged.
- **Negative withdrawal:** Attempt to withdraw a negative amount. The withdrawal should be denied, and the balance should remain unchanged.
- **Non-numeric deposit:** Attempt to deposit a non-numeric value (e.g., "abc"). The deposit should be denied, and the balance should remain unchanged.
- **Non-numeric withdrawal:** Attempt to withdraw a non-numeric value (e.g., "xyz"). The withdrawal should be denied, and the balance should remain unchanged.

#TASK-5

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

OUTPUT:



The screenshot shows a Google Colab notebook with a Python function `is_number_palindrome` and its test cases. The function checks if a number is a palindrome by converting it to a string and reversing it. It includes comments for arguments, returns, and test cases. The test cases are printed out, showing the function's output for various inputs.

```
[18] def is_number_palindrome(num):  
    """  
    Checks if an integer reads the same backward.  
  
    Args:  
        num: An integer.  
  
    Returns:  
        True if the number is a palindrome, False otherwise.  
    """  
    if not isinstance(num, int):  
        return False  
    if num < 0:  
        return False # Negative numbers are not considered palindromes  
  
    # Convert the number to a string to easily reverse it  
    num_str = str(num)  
    return num_str == num_str[::-1]  
  
    # Test cases for is_number_palindrome  
    print(f"is_number_palindrome(121): {is_number_palindrome(121)}")  
    print(f"is_number_palindrome(123): {is_number_palindrome(123)}")  
    print(f"is_number_palindrome(0): {is_number_palindrome(0)}")  
    print(f"is_number_palindrome(-121): {is_number_palindrome(-121)}")  
    print(f"is_number_palindrome(1): {is_number_palindrome(1)}")  
    print(f"is_number_palindrome(11): {is_number_palindrome(11)}")  
    print(f"is_number_palindrome(1221): {is_number_palindrome(1221)}")  
    print(f"is_number_palindrome(12345): {is_number_palindrome(12345)}")  
    print(f"is_number_palindrome('abc'): {is_number_palindrome('abc')}")  
    print(f"is_number_palindrome(None): {is_number_palindrome(None)}")  
  
is_number_palindrome(121): True  
is_number_palindrome(123): False  
is_number_palindrome(0): True  
is_number_palindrome(-121): False  
is_number_palindrome(1): True  
is_number_palindrome(11): True  
is_number_palindrome(1221): True  
is_number_palindrome(12345): False  
is_number_palindrome('abc'): False  
is_number_palindrome(None): False
```

Here are some test cases for the `is_number_palindrome(num)` function:

- `num = 121`: Should return `True` (reads the same backward)
- `num = 123`: Should return `False` (does not read the same backward)
- `num = 0`: Should return `True` (single digit is a palindrome)
- `num = -121`: Should return `False` (negative numbers are generally not considered palindromes)
- `num = 1`: Should return `True` (single digit)
- `num = 11`: Should return `True`
- `num = 1221`: Should return `True`
- `num = 12345`: Should return `False`
- `num = "abc"`: Should handle invalid input (e.g., return `False` or raise an error)
- `num = None`: Should handle invalid input

