

AI ASSISTED CODING

NAME: T.Akanksha

HT NO: 2403A510D8

BATCH NO:01

ASSIGNMENT 8.2

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:

Generate test cases for a function `is_prime(n)` and then implement the function. The function should return `True` if the input integer `n` is prime and `False` otherwise. Only integers greater than 1 can be prime. Include test cases for edge cases: 0, 1, 2, negative numbers, and large primes.

CODE:

```
▶ 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 <= 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  
  
# Generate test cases based on requirements and edge cases  
  
# Integers > 1  
print(f"Is 7 prime? {is_prime(7)}")  
print(f"Is 10 prime? {is_prime(10)}")  
  
# Edge cases  
print(f"Is 0 prime? {is_prime(0)}")  
print(f"Is 1 prime? {is_prime(1)}")  
print(f"Is 2 prime? {is_prime(2)}")  
print(f"Is -5 prime? {is_prime(-5)}")  
print(f"Is 1000003 prime? {is_prime(1000003)}") # A large prime  
print(f"Is 1000001 prime? {is_prime(1000001)}") # Not a large prime (1000001 = 101 * 9901)  
  
# Other cases  
print(f"Is 4 prime? {is_prime(4)}")  
print(f"Is 9 prime? {is_prime(9)}")
```

OUTPUT:

```
⇒ Is 7 prime? True
   Is 10 prime? False
   Is 0 prime? False
   Is 1 prime? False
   Is 2 prime? True
   Is -5 prime? False
   Is 1000003 prime? True
   Is 1000001 prime? False
   Is 4 prime? False
   Is 9 prime? False
```

EXPLANATION:

The first code cell defines the `is_prime` function, which efficiently checks if an integer is a prime number. The second code cell provides various test cases to verify the `is_prime` function's correctness, including edge cases like 0, 1, and negative numbers, as well as large prime and non-prime numbers.

TASK 2~

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 and implement functions for converting between Celsius and Fahrenheit. Include test cases for known pairs ($0^{\circ}\text{C} = 32^{\circ}\text{F}$, $100^{\circ}\text{C} = 212^{\circ}\text{F}$), decimals, and invalid inputs (strings, None).

CODE:

```
def celsius_to_fahrenheit(c):  
    """Converts Celsius to Fahrenheit."""  
    if not isinstance(c, (int, float)):  
        return None # Handle invalid input  
  
    return (c * 9/5) + 32  
  
def fahrenheit_to_celsius(f):  
    """Converts Fahrenheit to Celsius."""  
    if not isinstance(f, (int, float)):  
        return None # Handle invalid input  
  
    return (f - 32) * 5/9
```

```
# Generate test cases
```

```
# Known pairs
```

```
print(f"0°C is {celsius_to_fahrenheit(0)}°F")  
print(f"100°C is {celsius_to_fahrenheit(100)}°F")  
print(f"32°F is {fahrenheit_to_celsius(32)}°C")  
print(f"212°F is {fahrenheit_to_celsius(212)}°C")
```

```
# Decimals
```

```
print(f"25.5°C is {celsius_to_fahrenheit(25.5)}°F")  
print(f"77.9°F is {fahrenheit_to_celsius(77.9)}°C")
```

```
# Invalid inputs
```

```
print(f"'abc'°C is {celsius_to_fahrenheit('abc')}°F")  
print(f"None°C is {celsius_to_fahrenheit(None)}°F")  
print(f"'xyz'°F is {fahrenheit_to_celsius('xyz')}°C")  
print(f"None°F is {fahrenheit_to_celsius(None)}°C")
```

OUTPUT:

```
⇒ 0°C is 32.0°F  
100°C is 212.0°F  
32°F is 0.0°C  
212°F is 100.0°C  
25.5°C is 77.9°F  
77.9°F is 25.500000000000004°C  
'abc'°C is None°F  
None°C is None°F  
'xyz'°F is None°C  
None°F is None°C
```

EXPLANATION:

The first code cell defines two functions:

celsius_to_fahrenheit and fahrenheit_to_celsius for converting temperatures, including handling invalid inputs. The second code cell contains test cases for these functions, verifying known conversions, decimal values, and how invalid inputs are handled.

TASK 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 a function `count_words(text)` that returns the number of words in a sentence, handling normal text, multiple spaces, punctuation, and

empty strings. Then, implement the function and test it with the generated test cases.

CODE:

```
import string

def count_words(text):
    """
    Counts the number of words in a sentence.

    Args:
        text: A string representing the sentence.

    Returns:
        The number of words in the sentence.
    """
    if not isinstance(text, str):
        return 0 # Or raise an error, depending on desired behavior for non-string input

    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Split by whitespace and filter out empty strings
    words = text.split()
    return len(words)
```

```
# Generate test cases

# Normal text
print(f'{"This is a normal sentence."} has {count_words("This is a normal sentence.")} words.')

# Multiple spaces
print(f'{"This has multiple spaces."} has {count_words("This has multiple spaces.")} words.')

# Punctuation
print(f'{"This sentence has punctuation, right?"} has {count_words("This sentence has punctuation, right?")} words.')

# Empty string
print(f'{""} has {count_words("")} words.')

# String with only spaces
print(f'{"   "} has {count_words("   ")} words.')

# String with only punctuation
print(f'{".,!?"} has {count_words(".,!?")} words.')

# Mixed case and punctuation
print(f'{"Hello, World! How are you?"} has {count_words("Hello, World! How are you?")} words.')
```

OUTPUT:

```
⇒ 'This is a normal sentence.' has 5 words.  
  'This has multiple spaces.' has 4 words.  
  'This sentence has punctuation, right?' has 5 words.  
  '' has 0 words.  
  '   ' has 0 words.  
  '.,!?' has 0 words.  
  'Hello, World! How are you?' has 5 words.
```

EXPLANATION:

This code cell provides test cases for the `count_words` function. It calls the function with different types of strings, including:

- A normal sentence.
- A sentence with multiple spaces between words.
- A sentence with punctuation.
- An empty string.
- A string containing only spaces.
- A string containing only punctuation.
- A string with mixed case and punctuation.

For each test case, it prints the input string and the number of words returned by the `count_words` function, allowing you to verify that the function handles these different scenarios correctly.

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

PROMPT:

Generate test cases and implement a BankAccount class with deposit, withdraw, and check_balance methods, ensuring negative deposits/withdrawals raise errors and withdrawals do not exceed the balance.

CODE:

```
class BankAccount:
    """Represents a simple bank account."""

    def __init__(self, initial_balance=0):
        if not isinstance(initial_balance, (int, float)) or initial_balance < 0:
            raise ValueError("Initial balance must be a non-negative number.")
        self.balance = initial_balance

    def deposit(self, amount):
        """Deposits a positive amount into the account."""
        if not isinstance(amount, (int, float)) or amount <= 0:
            raise ValueError("Deposit amount must be a positive number.")
        self.balance += amount
        print(f"Deposited: ${amount}. New balance: ${self.balance}")

    def withdraw(self, amount):
        """Withdraws a positive amount from the account if sufficient funds are available."""
        if not isinstance(amount, (int, float)) or amount <= 0:
            raise ValueError("Withdrawal amount must be a positive number.")
        if amount > self.balance:
            raise ValueError("Insufficient funds.")
        self.balance -= amount
        print(f"Withdrew: ${amount}. New balance: ${self.balance}")

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




Generate test cases

Test case 1: Initial balance and check_balance

account1 = BankAccount(100)

print(f"Initial balance: \${account1.check_balance()}")

Test case 2: Deposit

account1.deposit(50)

print(f"Balance after deposit: \${account1.check_balance()}")

Test case 3: Successful withdrawal

account1.withdraw(30)

print(f"Balance after withdrawal: \${account1.check_balance()}")

Test case 4: Attempt to withdraw more than balance (should raise error)

try:

account1.withdraw(200)

except ValueError as e:

print(f"Withdrawal error: {e}")

Test case 5: Attempt negative deposit (should raise error)

try:

account1.deposit(-50)

except ValueError as e:

print(f"Deposit error: {e}")

Test case 6: Attempt negative withdrawal (should raise error)

try:

account1.withdraw(-20)

except ValueError as e:

print(f"Withdrawal error: {e}")

Test case 7: Initial balance with zero

account2 = BankAccount()

print(f"Initial balance of account2: \${account2.check_balance()}")

```

# Test case 8: Deposit zero (should raise error)
try:
    account2.deposit(0)
except ValueError as e:
    print(f"Deposit error: {e}")

# Test case 9: Withdraw zero (should raise error)
try:
    account2.withdraw(0)
except ValueError as e:
    print(f"Withdrawal error: {e}")

# Test case 10: Invalid initial balance (should raise error)
try:
    account3 = BankAccount("abc")
except ValueError as e:
    print(f"Initial balance error: {e}")

```

OUTPUT:

```

➡ Initial balance: $100
   Deposited: $50. New balance: $150
   Balance after deposit: $150
   Withdrew: $30. New balance: $120
   Balance after withdrawal: $120
   Withdrawal error: Insufficient funds.
   Deposit error: Deposit amount must be a positive number.
   Withdrawal error: Withdrawal amount must be a positive number.
   Initial balance of account2: $0
   Deposit error: Deposit amount must be a positive number.
   Withdrawal error: Withdrawal amount must be a positive number.
   Initial balance error: Initial balance must be a non-negative number.

```

EXPLANATION:

This code cell contains various test cases for the BankAccount class. It demonstrates how to:

- Create a BankAccount instance with an initial balance and check the balance.
- Deposit a positive amount.
- Withdraw a positive amount successfully.

- Handle errors for attempting to withdraw more than the balance, depositing a negative amount, withdrawing a negative amount, and attempting to deposit or withdraw zero.
- Handle errors for providing an invalid initial balance.

These test cases help verify that the BankAccount class behaves as expected under different scenarios, including error conditions.

TASK 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:

CODE:

```
▶ def is_number_palindrome(num):  
    """  
    Checks if an integer is a palindrome.  
  
    Args:  
        num: An integer.  
  
    Returns:  
        True if the number is a palindrome, False otherwise.  
    """  
    # Handle negative numbers and single-digit numbers (including 0)  
    if num < 0:  
        return False # Negative numbers are not palindromes  
    if 0 <= num < 10:  
        return True # Single-digit numbers are palindromes  
  
    # Convert the number to a string to easily reverse it  
    num_str = str(num)  
  
    # Check if the string is equal to its reverse  
    return num_str == num_str[::-1]
```

```
▶ # Generate test cases  
  
# Examples  
print(f"Is 121 a palindrome? {is_number_palindrome(121)}")  
print(f"Is 123 a palindrome? {is_number_palindrome(123)}")  
  
# Edge cases  
print(f"Is 0 a palindrome? {is_number_palindrome(0)}")  
print(f"Is -121 a palindrome? {is_number_palindrome(-121)}")  
print(f"Is 5 a palindrome? {is_number_palindrome(5)}")  
  
# Other cases  
print(f"Is 1221 a palindrome? {is_number_palindrome(1221)}")  
print(f"Is 12345 a palindrome? {is_number_palindrome(12345)}")  
print(f"Is 1001 a palindrome? {is_number_palindrome(1001)}")  
print(f"Is 10 a palindrome? {is_number_palindrome(10)}")
```

OUTPUT:

```
⇒ Is 121 a palindrome? True  
Is 123 a palindrome? False  
Is 0 a palindrome? True  
Is -121 a palindrome? False  
Is 5 a palindrome? True  
Is 1221 a palindrome? True  
Is 12345 a palindrome? False  
Is 1001 a palindrome? True  
Is 10 a palindrome? False
```

EXPLANATION:

This code cell contains test cases for the `is_number_palindrome` function. It calls the function with various integer inputs, including:

- The examples provided (121 and 123).
- Edge cases like 0, a negative number (-121), and a single-digit positive number (5).
- Other numbers that are palindromes (1221, 1001) and not palindromes (12345, 10).

For each input, it prints whether the number is a palindrome based on the function's output, allowing you to verify the function's behavior for different scenarios.