AI ASSISTED CODING

NAME : NAREDDY JASMITHA

ENROLLMENT NO : 2403A51281

BRANCH :CSE/AIML

COURSE : AI

SEM : 3RD

DATE : 23-09-2025

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 a python code to develop 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

Expected Output#1

• A working prime checker that passes AI-generated tests using edge coverage

```
# Test cases for the is_prime function
test_cases = [
    (0, False),  # Edge case: 0
    (1, False),  # Edge case: 1
    (2, True),   # Edge case: 2 (first prime)
    (3, True),
    (4, False),
    (5, True),
    (10, False),
    (17, True),
    (-5, False), # Edge case: negative number
    (97, True),  # Large prime
    (100, False),
    (997, True), # Another large prime
]
```

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

    Args:
        n: An integer.

    Returns:
        True if n is prime, False otherwise.
    """
    # Requirement: Only integers > 1 can be prime.
    if not isinstance(n, int) or n <= 1:
        return False

    # Edge case: 2 is the only even prime number
    if n == 2:
        return True

    # Eliminate even numbers greater than 2
    if n % 2 == 0:
        return False

    # Check for divisibility from 3 up to the square root of n, skipping even numbers
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2

    return True
```

```python
# Run the test cases
for number, expected in test_cases:
    result = is_prime(number)
    assert result == expected, f"Input: {number}, Expected: {expected}, Got: {result}"
    print(f"Test case {number}: {'Passed' if result == expected else 'Failed'}")

print("\nAll test cases passed!")
```

```
Test case 0: Passed
Test case 1: Passed
Test case 2: Passed
Test case 3: Passed
Test case 4: Passed
Test case 5: Passed
Test case 10: Passed
Test case 17: Passed
Test case -5: Passed
Test case 97: Passed
Test case 100: Passed
Test case 997: Passed

All test cases passed!
```

EXPLANATION:

This code cell defines a list of `test_cases`. Each element in the list is tuple containing a number and the expected boolean result (`True` if t number is prime, `False` otherwise) when passed to the `is_prime` function. These test cases cover various scenarios, including edge cas like 0, 1, 2, negative numbers, and both small and large prime and non-prime numbers.

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°C = 32°F, 100°C = 212°F.

• Include decimals and invalid inputs like strings or None

PROMPT:

Generate a python code to develop test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). Requirements • Validate known pairs: 0°C = 32°F, 100°C = 212°F. • Include decimals and invalid inputs like strings or None

Expected Output#2

Dual conversion functions with complete test coverage and safe type handling

```python
# Test cases for celsius_to_fahrenheit and fahrenheit_to_celsius
temp_test_cases = [
    (0, 32),      # 0°C = 32°F
    (100, 212),   # 100°C = 212°F
    (25.5, 77.9), # Decimal
    (-10, 14),    # Negative temperature
    ('abc', None), # Invalid input: string
    (None, None)  # Invalid input: None
]
```

EXPLANATION:

This code cell defines a list called `temp_test_cases`. Each element in the list is a tuple representing a pair of temperatures: the first value is a temperature in Celsius, and the second value is the expected equivalent in Fahrenheit. These test cases are designed to verify the correctness of functions that convert between Celsius and Fahrenheit, including known values, decimals, negative temperatures, and invalid inputs like strings or `None`.

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 a python code to develop 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

Expected Output#3
Accurate word count with robust test case validation.

```python
# Test cases for the count_words function
count_words_test_cases = [
    ("Hello world", 2),  # Normal text
    ("  Leading and trailing spaces  ", 5),  # Leading and trailing spaces
    ("Multiple   spaces", 2),  # Multiple spaces between words
    ("Punctuation, like this!", 3),  # Punctuation
    ("", 0),  # Empty string
    ("   ", 0),  # String with only spaces
    ("One", 1),  # Single word
    ("Word with-hyphen", 2), # Hyphenated word can be tricky
    ("Word with'apostrophe", 2), # Apostrophe can be tricky
    ("Hello.World", 2) # Punctuation without space
]
```

EXPLANATION:

This code cell defines a list of `count_words_test_cases`. Each element in the list is a tuple where the first item is a string of text and the second item is the expected number of words in that string. These test cases are designed to verify the behavior of a `count_words` function under various conditions, including normal sentences, strings with leading/trailing spaces, multiple spaces between words, punctuation, empty strings, strings with only spaces, single words, and words with hyphens or apostrophes.

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 a python code to develop test cases for a Bank Account class with: Methods: deposit(amount) withdraw(amount) check_balance() Requirements: • Negative deposits/withdrawals should raise an error. • Cannot withdraw more than balance

Expected Output#4

• AI-generated test suite with a robust class that handles all test cases.

```
# Test cases for the BankAccount class
bank_account_test_cases = [
    # Test deposit
    ({"initial_balance": 100, "actions": [("deposit", 50)], "expected_balance": 150, "expected_error": Non
    ({"initial_balance": 100, "actions": [("deposit", -50)], "expected_balance": 100, "expected_error": Va

    # Test withdraw
    ({"initial_balance": 100, "actions": [("withdraw", 50)], "expected_balance": 50, "expected_error": Non
    ({"initial_balance": 100, "actions": [("withdraw", 150)], "expected_balance": 100, "expected_error": V
    ({"initial_balance": 100, "actions": [("withdraw", -50)], "expected_balance": 100, "expected_error": V

    # Test multiple actions
    ({"initial_balance": 100, "actions": [("deposit", 50), ("withdraw", 25)], "expected_balance": 125, "ex
    ({"initial_balance": 100, "actions": [("deposit", 50), ("withdraw", 200)], "expected_balance": 150, "e
]
```

EXPLANATION:

This code cell defines a list called `bank_account_test_cases`. Each element in the list is a dictionary that describes a test scenario for a `BankAccount` class. The dictionaries include:

- `"initial_balance"`: The starting balance of the account for the test.
- `"actions"`: A list of actions to perform on the account. Each action is a tuple with the method name (e.g., "deposit", "withdraw") and the amount.
- `"expected_balance"`: The expected final balance after performing the actions.
- `"expected_error"`: The type of error expected (if any) during the execution of the actions. This is used to test for invalid operations like negative deposits/withdrawals or withdrawing more than the balance.

These test cases are designed to cover various scenarios, including successful deposits and withdrawals, attempts to perform invalid

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

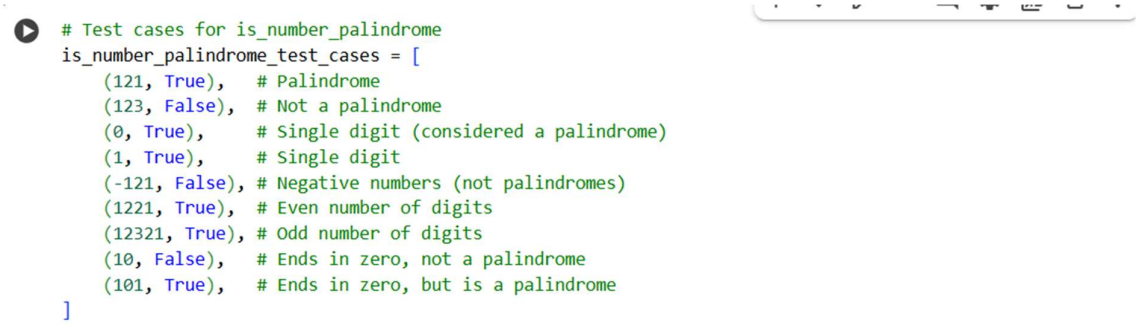
0, negative numbers → handled gracefully

PROMPT:

Generate a python code to develop test cases for is_number_palindrome(num), which checks if an integer reads the same backward.

Expected Output#5

- **Number-based palindrome checker function validated against test cases**



```
# Test cases for is_number_palindrome
is_number_palindrome_test_cases = [
    (121, True),   # Palindrome
    (123, False),  # Not a palindrome
    (0, True),     # Single digit (considered a palindrome)
    (1, True),     # Single digit
    (-121, False), # Negative numbers (not palindromes)
    (1221, True),  # Even number of digits
    (12321, True), # Odd number of digits
    (10, False),   # Ends in zero, not a palindrome
    (101, True),   # Ends in zero, but is a palindrome
]
```

EXPLANATION:

This code cell defines a list called `is_number_palindrome_test_cases`. Each element in the list is a tuple containing an integer and the expected boolean result (`True` if the number is a palindrome, `False` otherwise). These test cases cover various scenarios for checking if a number is a palindrome, including:

- Positive palindromes with odd and even numbers of digits.
- Positive numbers that are not palindromes.
- Single-digit numbers (which are considered palindromes).
- Negative numbers (which are generally not considered palindromes).
- Numbers ending in zero.

These tests will help verify that the `is_number_palindrome` function correctly identifies palindromic integers.