SR UNIVERSITY

AI ASSISTED CODING

ASSIGNMENT - 08

Name: Meer Burhan Ali Hashmi

Htno: 2503A51L44

Batch :20

Lab Objectives:

- To introduce students to test-driven development (TDD) using AI code generation tools.
- To enable the generation of test cases before writing code implementations.
- To reinforce the importance of testing, validation, and error handling.
- To encourage writing clean and reliable code based on AI-generated test expectations.

Lab Outcomes (LOs):

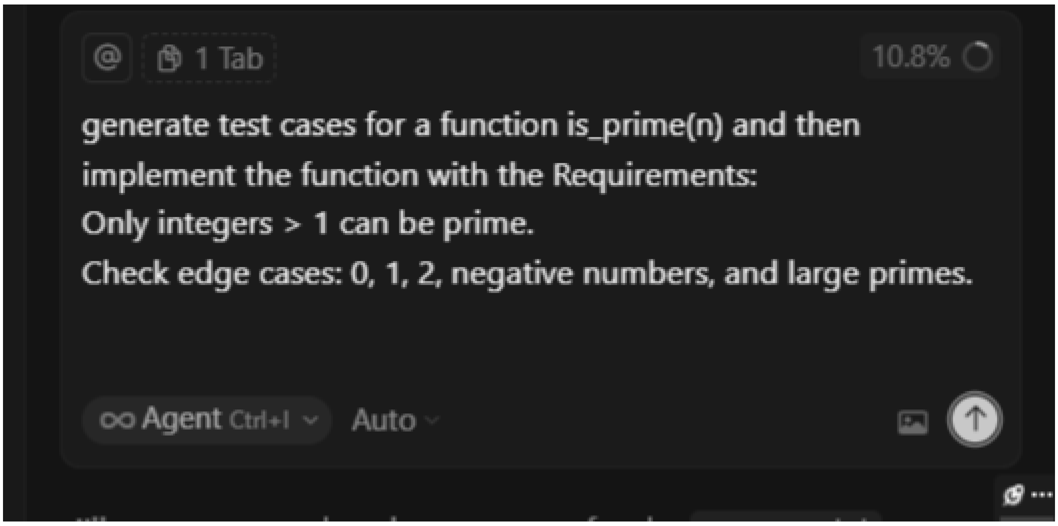
After completing this lab, students will be able to:

- Use AI tools to write test cases for Python functions and classes.
- Implement functions based on test cases in a test-first development style.
- Use unittest or pytest to validate code correctness.
- Analyze the completeness and coverage of AI-generated tests.
- Compare AI-generated and manually written test cases for quality and logic

Task Description#1

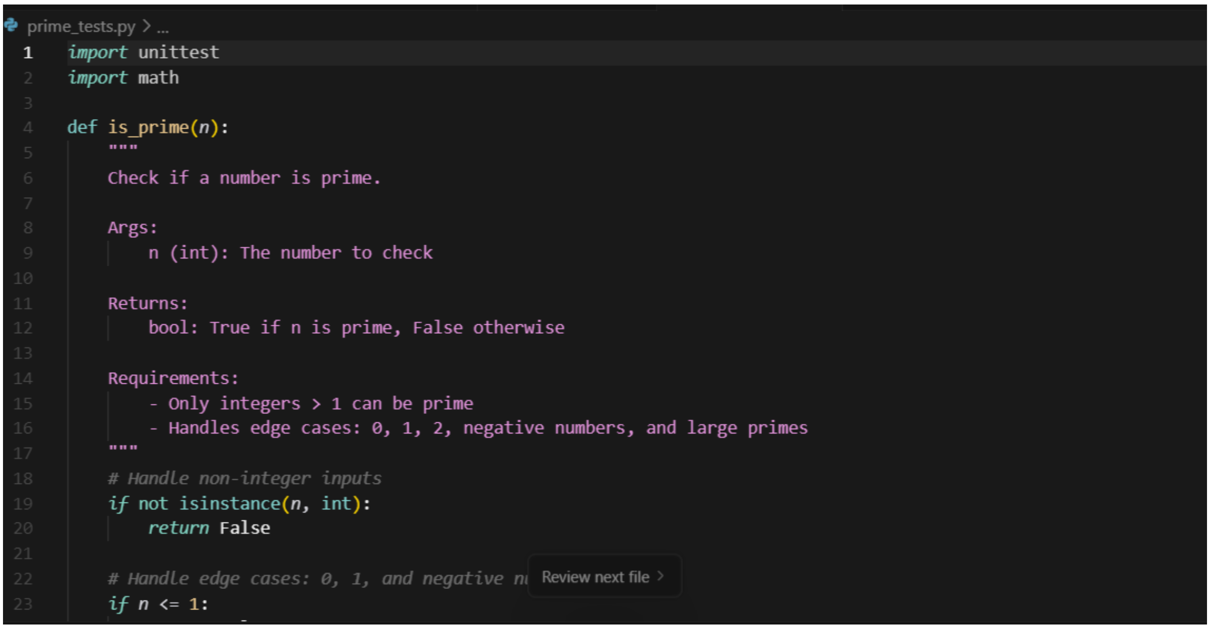
- Use AI to generate test cases for a function is_prime(n) and then implement the function.
- **Requirements:**
- Only integers > 1 can be prime.
- Check edge cases: 0, 1, 2, negative numbers, and large primes.

Entered prompt into the AI:

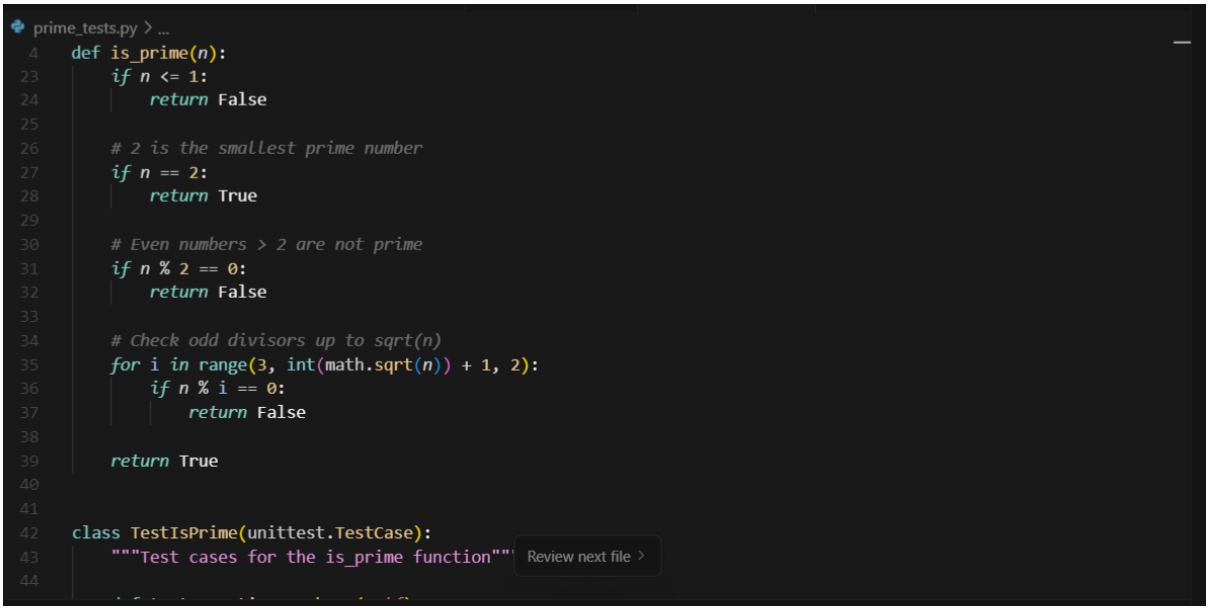


generate test cases for a function is_prime(n) and then
implement the function with the Requirements:
Only integers > 1 can be prime.
Check edge cases: 0, 1, 2, negative numbers, and large primes.

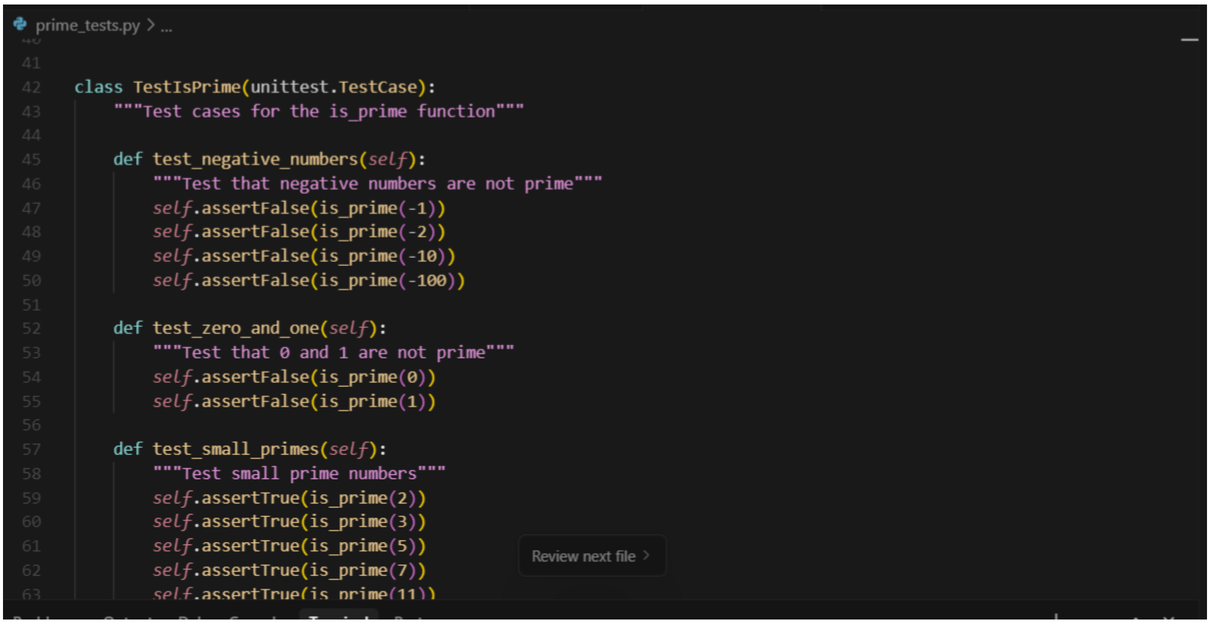
Given program by the AI:



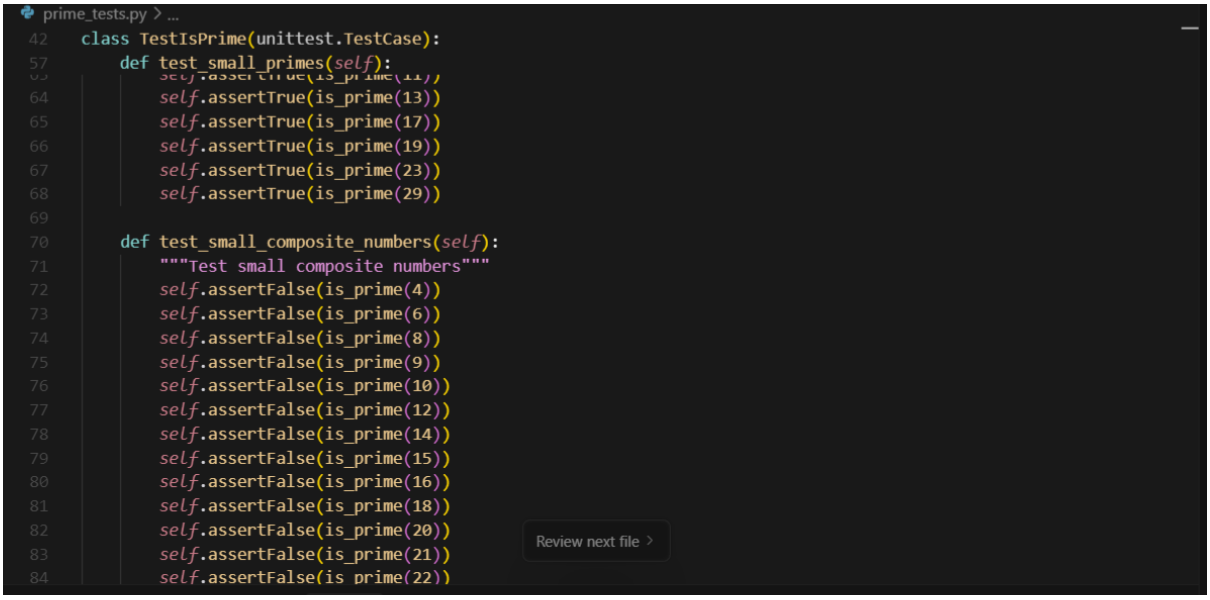
```python
import unittest
import math

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

    Args:
        n (int): The number to check

    Returns:
        bool: True if n is prime, False otherwise

    Requirements:
        - Only integers > 1 can be prime
        - Handles edge cases: 0, 1, 2, negative numbers, and large primes
    """
    # Handle non-integer inputs
    if not isinstance(n, int):
        return False

    # Handle edge cases: 0, 1, and negative nu
    if n <= 1:
```



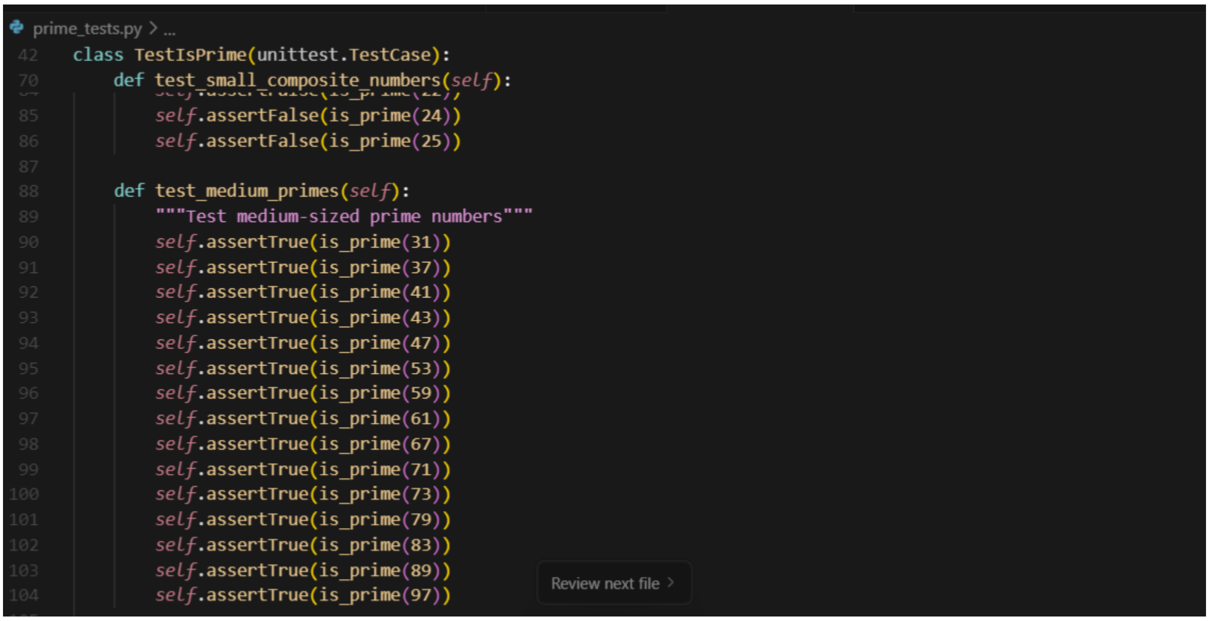
```python
def is_prime(n):
    if n <= 1:
        return False

    # 2 is the smallest prime number
    if n == 2:
        return True

    # Even numbers > 2 are not prime
    if n % 2 == 0:
        return False

    # Check odd divisors up to sqrt(n)
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False

    return True


class TestIsPrime(unittest.TestCase):
    """Test cases for the is_prime function"""

```



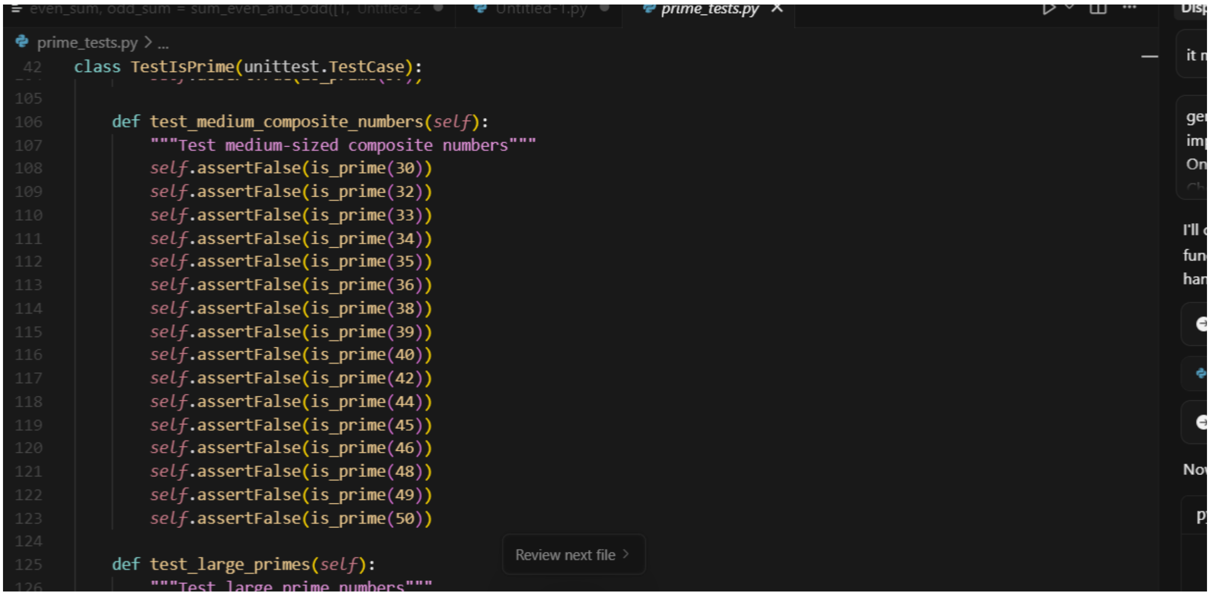
```python

class TestIsPrime(unittest.TestCase):
    """Test cases for the is_prime function"""

    def test_negative_numbers(self):
        """Test that negative numbers are not prime"""
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-2))
        self.assertFalse(is_prime(-10))
        self.assertFalse(is_prime(-100))

    def test_zero_and_one(self):
        """Test that 0 and 1 are not prime"""
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))

    def test_small_primes(self):
        """Test small prime numbers"""
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
        self.assertTrue(is_prime(5))
        self.assertTrue(is_prime(7))
        self.assertTrue(is_prime(11))
```



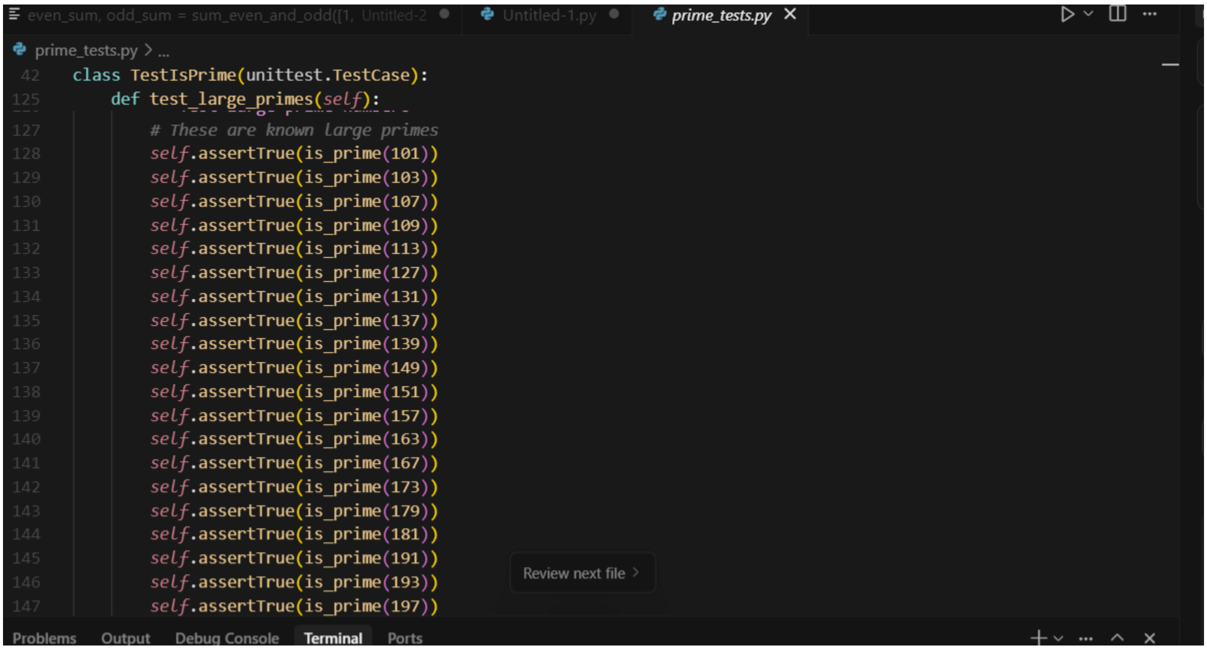
```
class TestIsPrime(unittest.TestCase):
    def test_small_primes(self):
        self.assertTrue(is_prime(13))
        self.assertTrue(is_prime(17))
        self.assertTrue(is_prime(19))
        self.assertTrue(is_prime(23))
        self.assertTrue(is_prime(29))

    def test_small_composite_numbers(self):
        """Test small composite numbers"""
        self.assertFalse(is_prime(4))
        self.assertFalse(is_prime(6))
        self.assertFalse(is_prime(8))
        self.assertFalse(is_prime(9))
        self.assertFalse(is_prime(10))
        self.assertFalse(is_prime(12))
        self.assertFalse(is_prime(14))
        self.assertFalse(is_prime(15))
        self.assertFalse(is_prime(16))
        self.assertFalse(is_prime(18))
        self.assertFalse(is_prime(20))
        self.assertFalse(is_prime(21))
        self.assertFalse(is_prime(22))
```



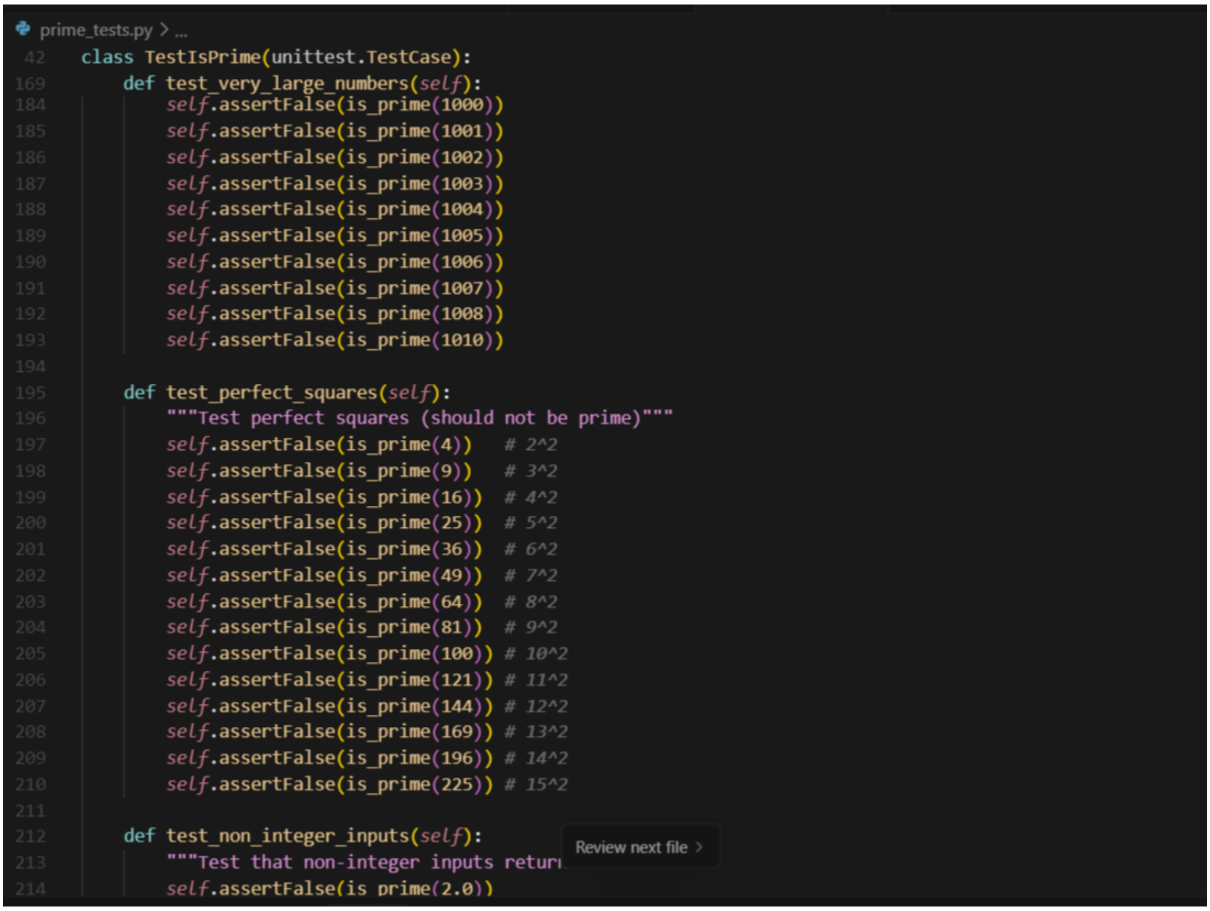
```
class TestIsPrime(unittest.TestCase):
    def test_small_composite_numbers(self):
        self.assertFalse(is_prime(24))
        self.assertFalse(is_prime(25))

    def test_medium_primes(self):
        """Test medium-sized prime numbers"""
        self.assertTrue(is_prime(31))
        self.assertTrue(is_prime(37))
        self.assertTrue(is_prime(41))
        self.assertTrue(is_prime(43))
        self.assertTrue(is_prime(47))
        self.assertTrue(is_prime(53))
        self.assertTrue(is_prime(59))
        self.assertTrue(is_prime(61))
        self.assertTrue(is_prime(67))
        self.assertTrue(is_prime(71))
        self.assertTrue(is_prime(73))
        self.assertTrue(is_prime(79))
        self.assertTrue(is_prime(83))
        self.assertTrue(is_prime(89))
        self.assertTrue(is_prime(97))
```



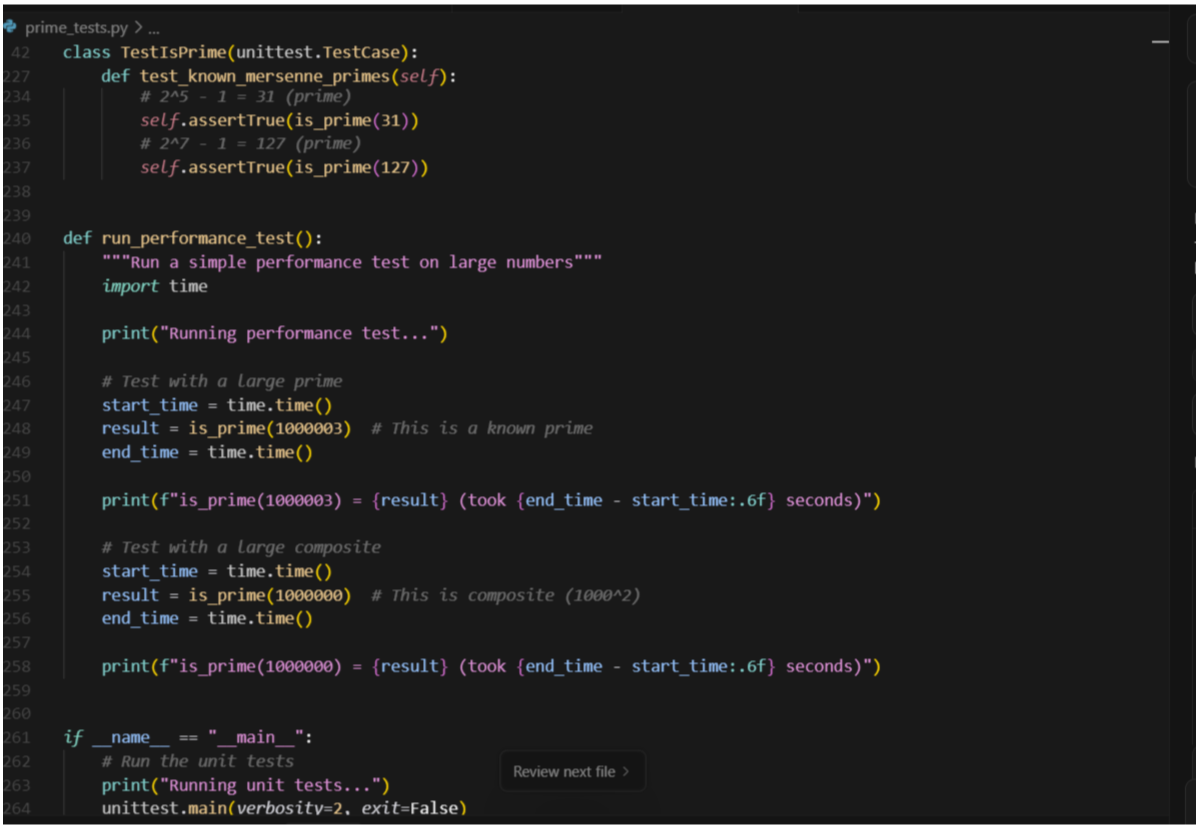
```
class TestIsPrime(unittest.TestCase):

    def test_medium_composite_numbers(self):
        """Test medium-sized composite numbers"""
        self.assertFalse(is_prime(30))
        self.assertFalse(is_prime(32))
        self.assertFalse(is_prime(33))
        self.assertFalse(is_prime(34))
        self.assertFalse(is_prime(35))
        self.assertFalse(is_prime(36))
        self.assertFalse(is_prime(38))
        self.assertFalse(is_prime(39))
        self.assertFalse(is_prime(40))
        self.assertFalse(is_prime(42))
        self.assertFalse(is_prime(44))
        self.assertFalse(is_prime(45))
        self.assertFalse(is_prime(46))
        self.assertFalse(is_prime(48))
        self.assertFalse(is_prime(49))
        self.assertFalse(is_prime(50))

    def test_large_primes(self):
```



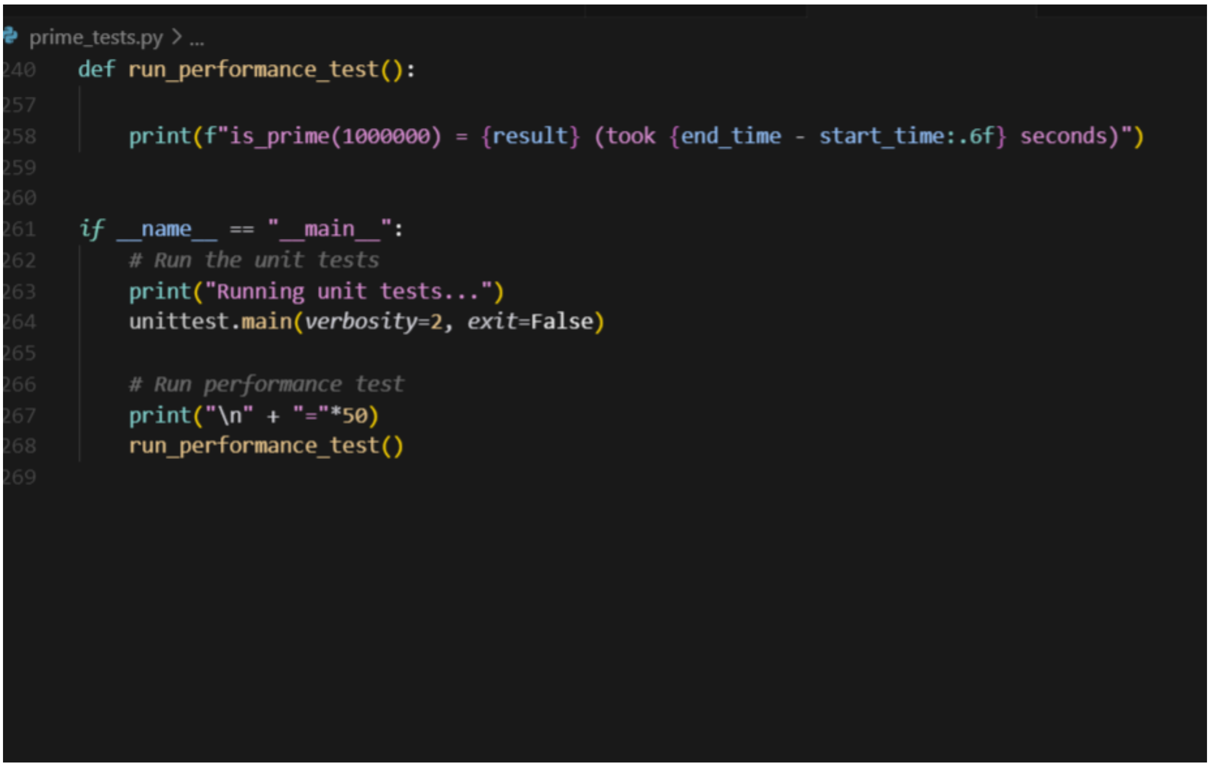
```python
class TestIsPrime(unittest.TestCase):
    def test_large_primes(self):
        # These are known large primes
        self.assertTrue(is_prime(101))
        self.assertTrue(is_prime(103))
        self.assertTrue(is_prime(107))
        self.assertTrue(is_prime(109))
        self.assertTrue(is_prime(113))
        self.assertTrue(is_prime(127))
        self.assertTrue(is_prime(131))
        self.assertTrue(is_prime(137))
        self.assertTrue(is_prime(139))
        self.assertTrue(is_prime(149))
        self.assertTrue(is_prime(151))
        self.assertTrue(is_prime(157))
        self.assertTrue(is_prime(163))
        self.assertTrue(is_prime(167))
        self.assertTrue(is_prime(173))
        self.assertTrue(is_prime(179))
        self.assertTrue(is_prime(181))
        self.assertTrue(is_prime(191))
        self.assertTrue(is_prime(193))
        self.assertTrue(is_prime(197))
```



```python
class TestIsPrime(unittest.TestCase):
    def test_very_large_numbers(self):
        self.assertFalse(is_prime(1000))
        self.assertFalse(is_prime(1001))
        self.assertFalse(is_prime(1002))
        self.assertFalse(is_prime(1003))
        self.assertFalse(is_prime(1004))
        self.assertFalse(is_prime(1005))
        self.assertFalse(is_prime(1006))
        self.assertFalse(is_prime(1007))
        self.assertFalse(is_prime(1008))
        self.assertFalse(is_prime(1010))

    def test_perfect_squares(self):
        """Test perfect squares (should not be prime)"""
        self.assertFalse(is_prime(4))   # 2^2
        self.assertFalse(is_prime(9))   # 3^2
        self.assertFalse(is_prime(16))  # 4^2
        self.assertFalse(is_prime(25))  # 5^2
        self.assertFalse(is_prime(36))  # 6^2
        self.assertFalse(is_prime(49))  # 7^2
        self.assertFalse(is_prime(64))  # 8^2
        self.assertFalse(is_prime(81))  # 9^2
        self.assertFalse(is_prime(100)) # 10^2
        self.assertFalse(is_prime(121)) # 11^2
        self.assertFalse(is_prime(144)) # 12^2
        self.assertFalse(is_prime(169)) # 13^2
        self.assertFalse(is_prime(196)) # 14^2
        self.assertFalse(is_prime(225)) # 15^2

    def test_non_integer_inputs(self):
        """Test that non-integer inputs retur
        self.assertFalse(is_prime(2.0))
```

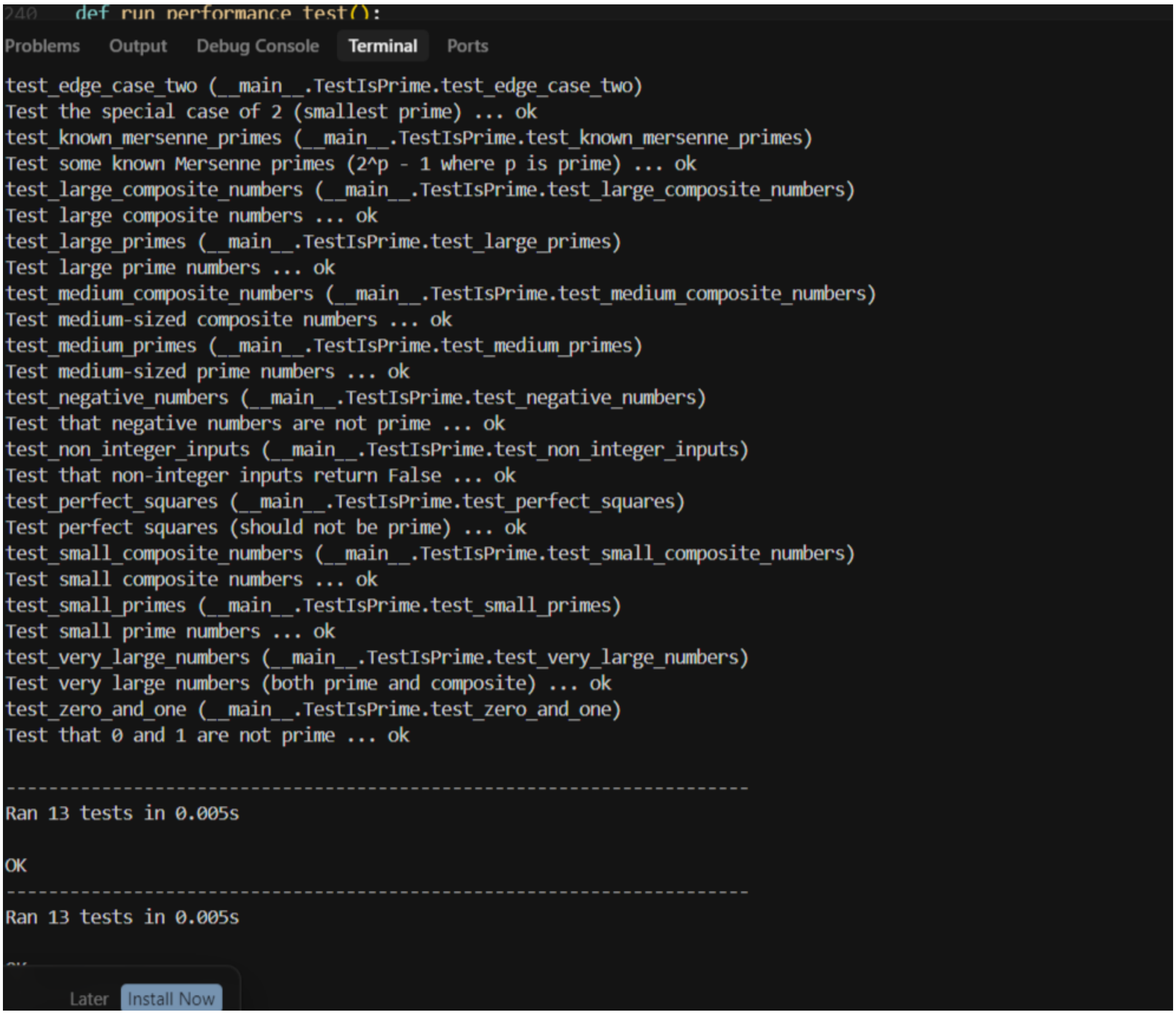


```
class TestIsPrime(unittest.TestCase):
    def test_known_mersenne_primes(self):
        # 2^5 - 1 = 31 (prime)
        self.assertTrue(is_prime(31))
        # 2^7 - 1 = 127 (prime)
        self.assertTrue(is_prime(127))


def run_performance_test():
    """Run a simple performance test on large numbers"""
    import time

    print("Running performance test...")

    # Test with a large prime
    start_time = time.time()
    result = is_prime(1000003)  # This is a known prime
    end_time = time.time()

    print(f"is_prime(1000003) = {result} (took {end_time - start_time:.6f} seconds)")

    # Test with a large composite
    start_time = time.time()
    result = is_prime(1000000)  # This is composite (1000^2)
    end_time = time.time()

    print(f"is_prime(1000000) = {result} (took {end_time - start_time:.6f} seconds)")


if __name__ == "__main__":
    # Run the unit tests
    print("Running unit tests...")
    unittest.main(verbosity=2, exit=False)
```



```
def run_performance_test():

    print(f"is_prime(1000000) = {result} (took {end_time - start_time:.6f} seconds)")


if __name__ == "__main__":
    # Run the unit tests
    print("Running unit tests...")
    unittest.main(verbosity=2, exit=False)

    # Run performance test
    print("\n" + "="*50)
    run_performance_test()
```

Output:



```
def run_performance_test():
```

```
test_edge_case_two (__main__.TestIsPrime.test_edge_case_two)
Test the special case of 2 (smallest prime) ... ok
test_known_mersenne_primes (__main__.TestIsPrime.test_known_mersenne_primes)
Test some known Mersenne primes (2^p - 1 where p is prime) ... ok
test_large_composite_numbers (__main__.TestIsPrime.test_large_composite_numbers)
Test large composite numbers ... ok
test_large_primes (__main__.TestIsPrime.test_large_primes)
Test large prime numbers ... ok
test_medium_composite_numbers (__main__.TestIsPrime.test_medium_composite_numbers)
Test medium-sized composite numbers ... ok
test_medium_primes (__main__.TestIsPrime.test_medium_primes)
Test medium-sized prime numbers ... ok
test_negative_numbers (__main__.TestIsPrime.test_negative_numbers)
Test that negative numbers are not prime ... ok
test_non_integer_inputs (__main__.TestIsPrime.test_non_integer_inputs)
Test that non-integer inputs return False ... ok
test_perfect_squares (__main__.TestIsPrime.test_perfect_squares)
Test perfect squares (should not be prime) ... ok
test_small_composite_numbers (__main__.TestIsPrime.test_small_composite_numbers)
Test small composite numbers ... ok
test_small_primes (__main__.TestIsPrime.test_small_primes)
Test small prime numbers ... ok
test_very_large_numbers (__main__.TestIsPrime.test_very_large_numbers)
Test very large numbers (both prime and composite) ... ok
test_zero_and_one (__main__.TestIsPrime.test_zero_and_one)
Test that 0 and 1 are not prime ... ok

----------------------------------------------------------------------
Ran 13 tests in 0.005s

OK
----------------------------------------------------------------------
Ran 13 tests in 0.005s
```

Observation:

The output displays the results of a Python script named def run_performance_test(). The script is likely a unit test suite for a function that determines if a number is prime. The output shows that all **13 tests** passed successfully in **0.005 seconds**.

Task Description#2 (Loops)

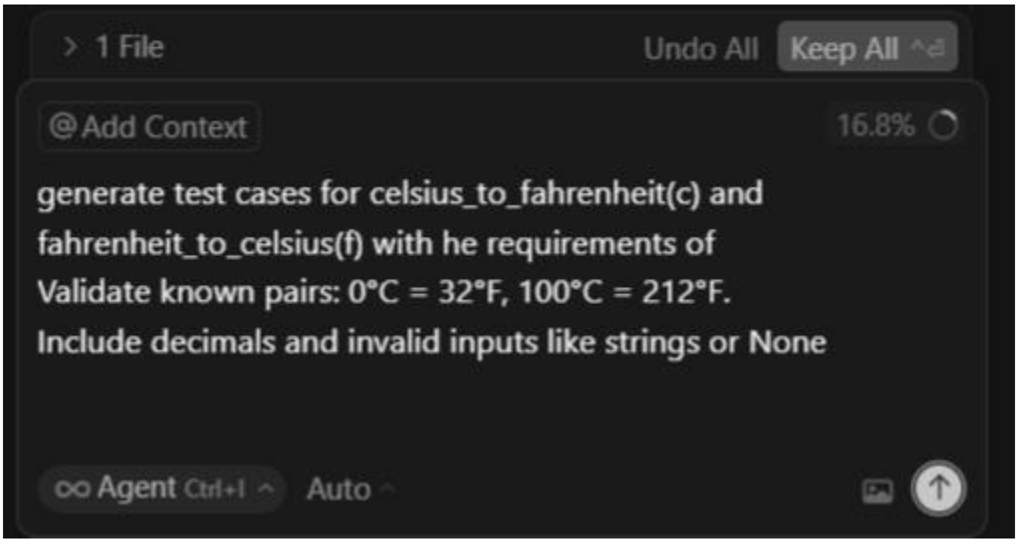
- Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f).

Requirements

- Validate known pairs: 0°C = 32°F, 100°C = 212°F.

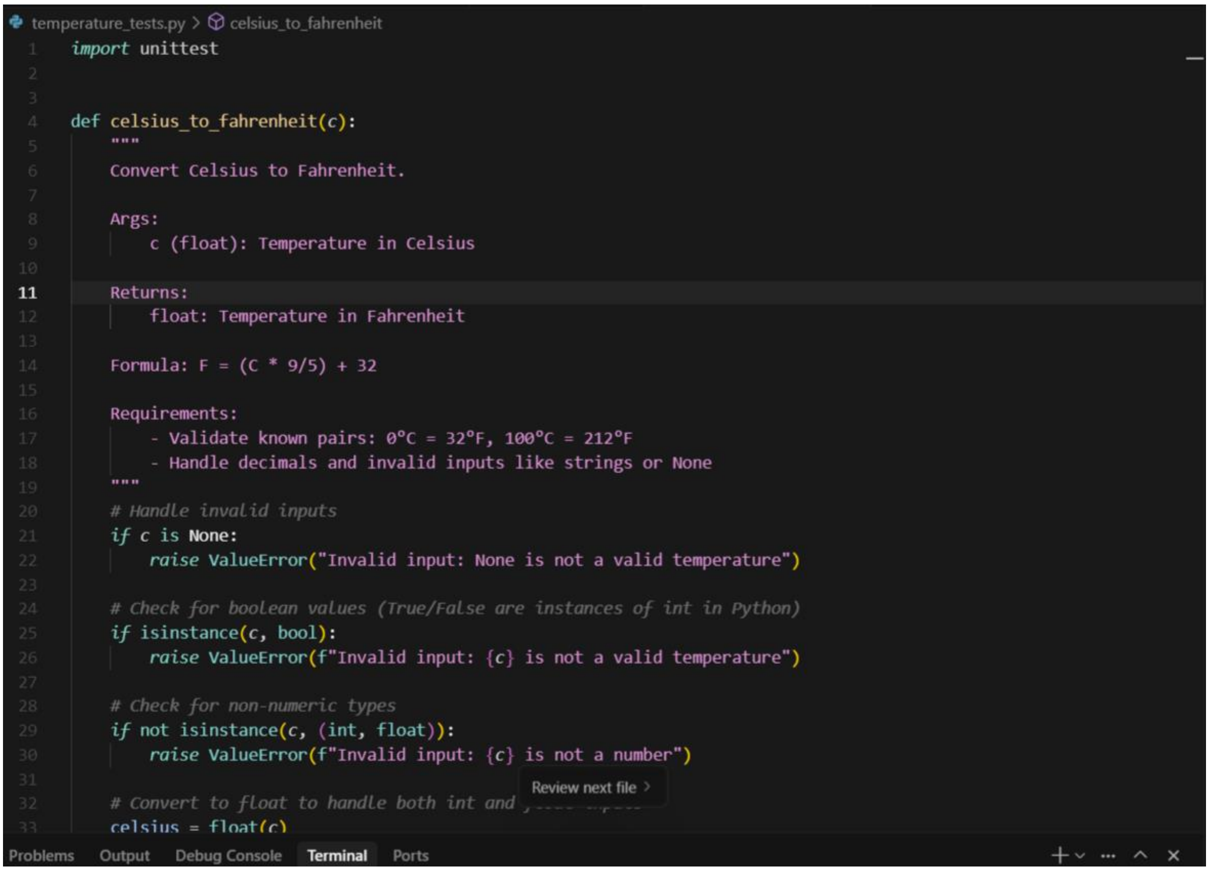
- Include decimals and invalid inputs like strings or None

Given prompt to the AI:

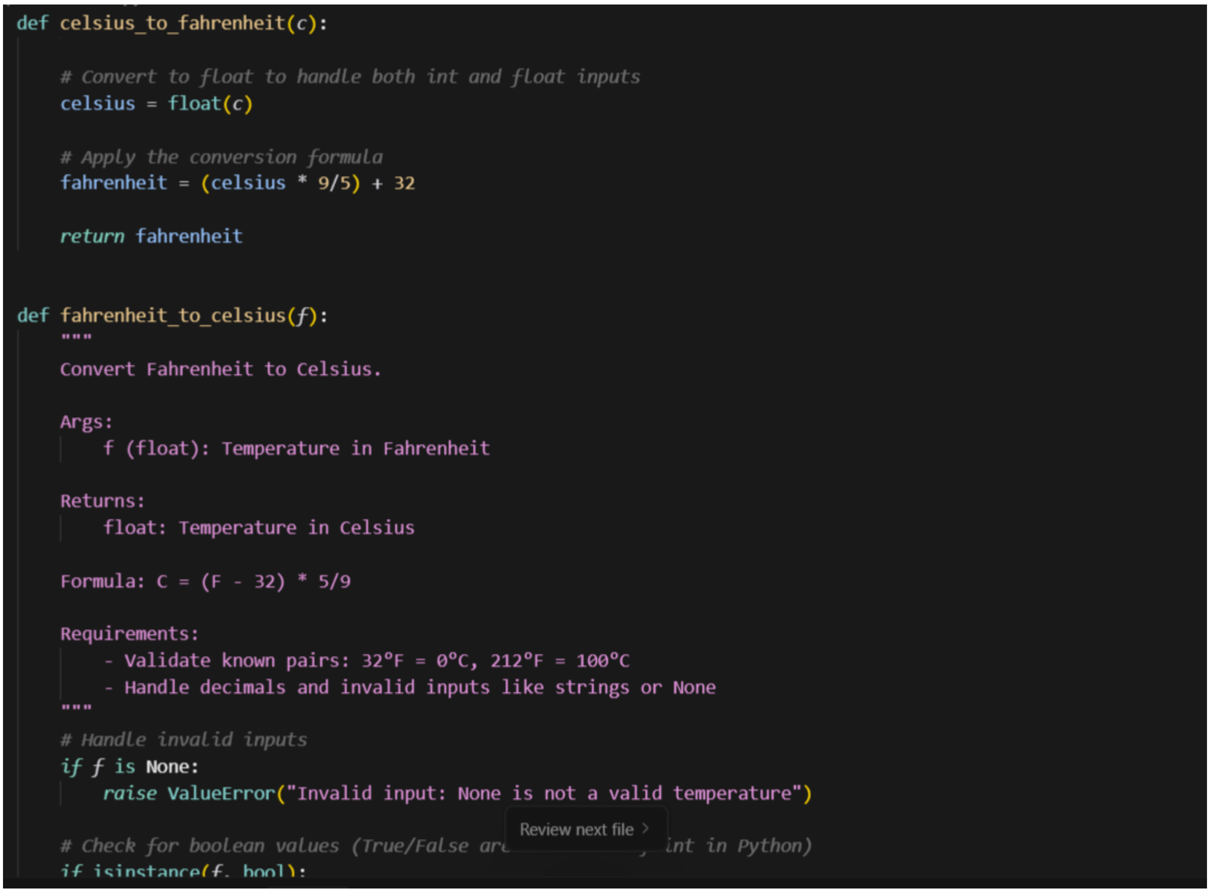


generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f) with he requirements of Validate known pairs: 0°C = 32°F, 100°C = 212°F. Include decimals and invalid inputs like strings or None

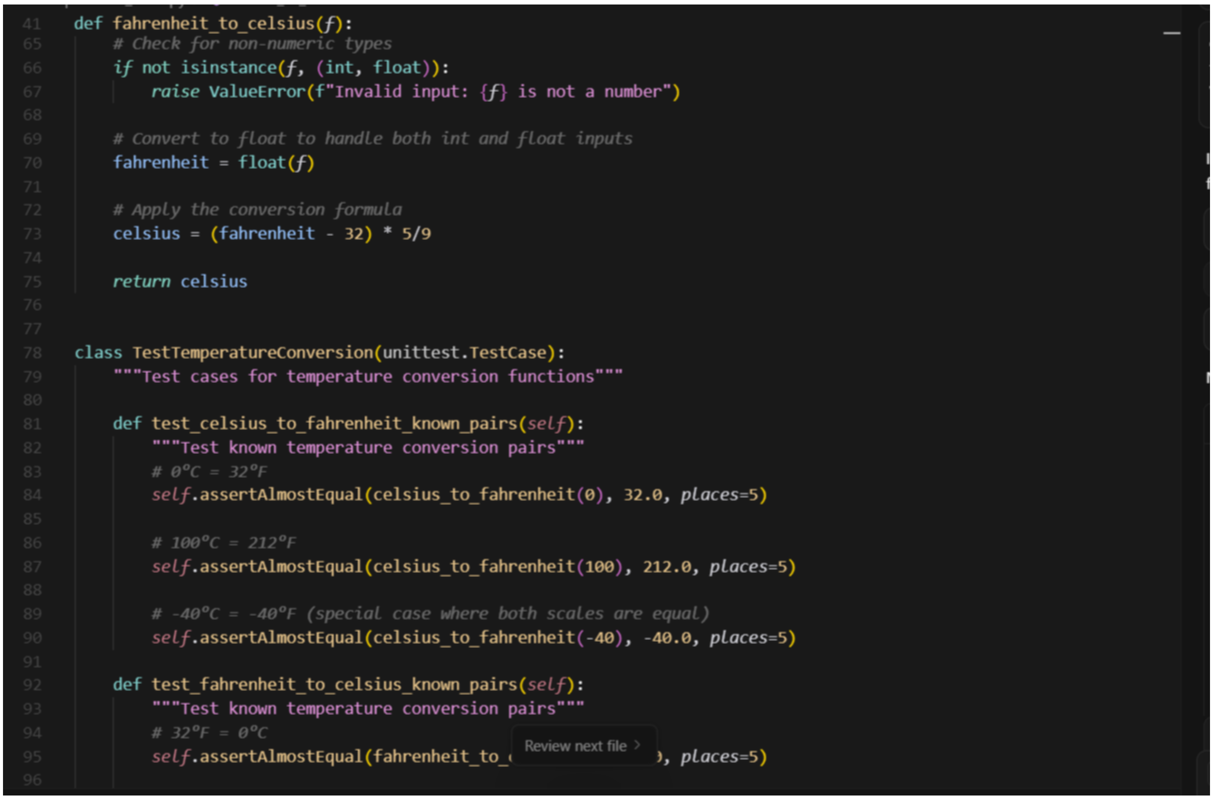
Given program by the AI:



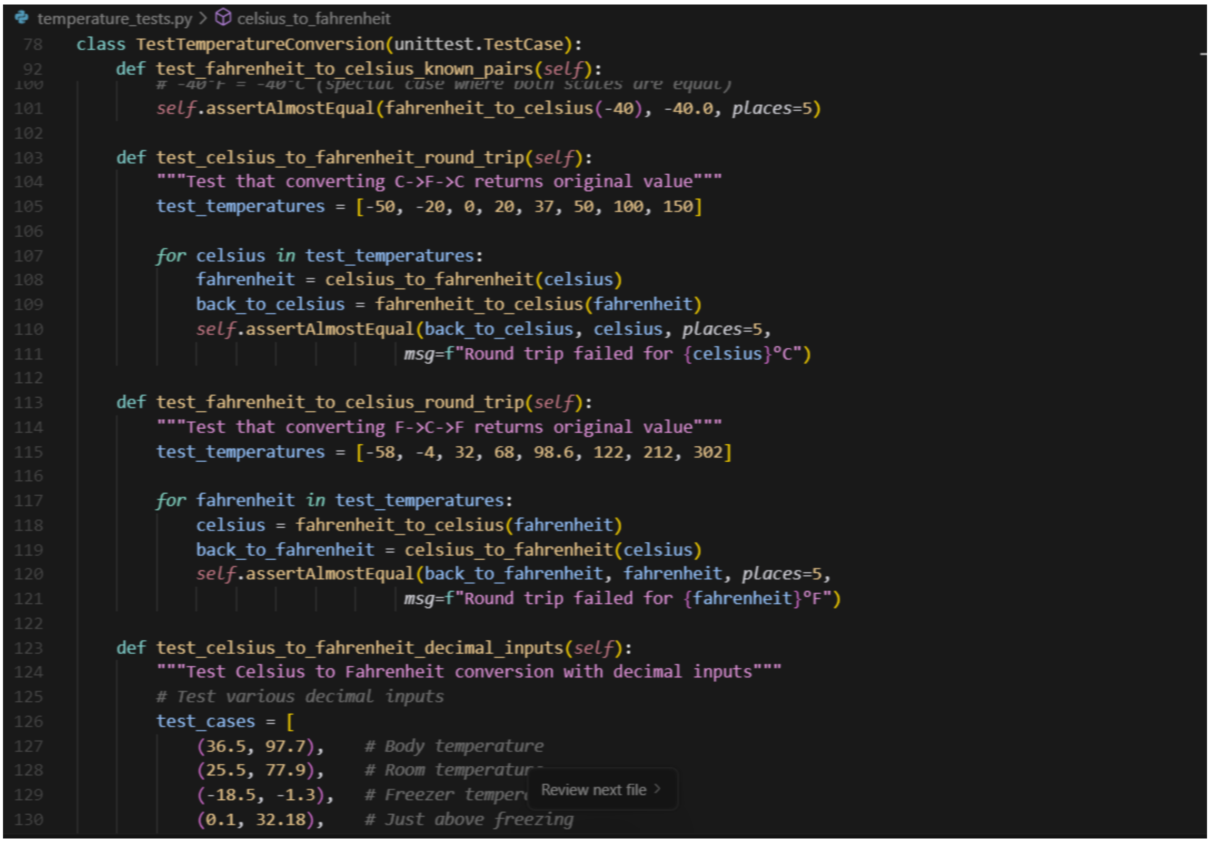
```python
import unittest


def celsius_to_fahrenheit(c):
    """
    Convert Celsius to Fahrenheit.

    Args:
        c (float): Temperature in Celsius

    Returns:
        float: Temperature in Fahrenheit

    Formula: F = (C * 9/5) + 32

    Requirements:
        - Validate known pairs: 0°C = 32°F, 100°C = 212°F
        - Handle decimals and invalid inputs like strings or None
    """
    # Handle invalid inputs
    if c is None:
        raise ValueError("Invalid input: None is not a valid temperature")

    # Check for boolean values (True/False are instances of int in Python)
    if isinstance(c, bool):
        raise ValueError(f"Invalid input: {c} is not a valid temperature")

    # Check for non-numeric types
    if not isinstance(c, (int, float)):
        raise ValueError(f"Invalid input: {c} is not a number")

    # Convert to float to handle both int and
    celsius = float(c)
```



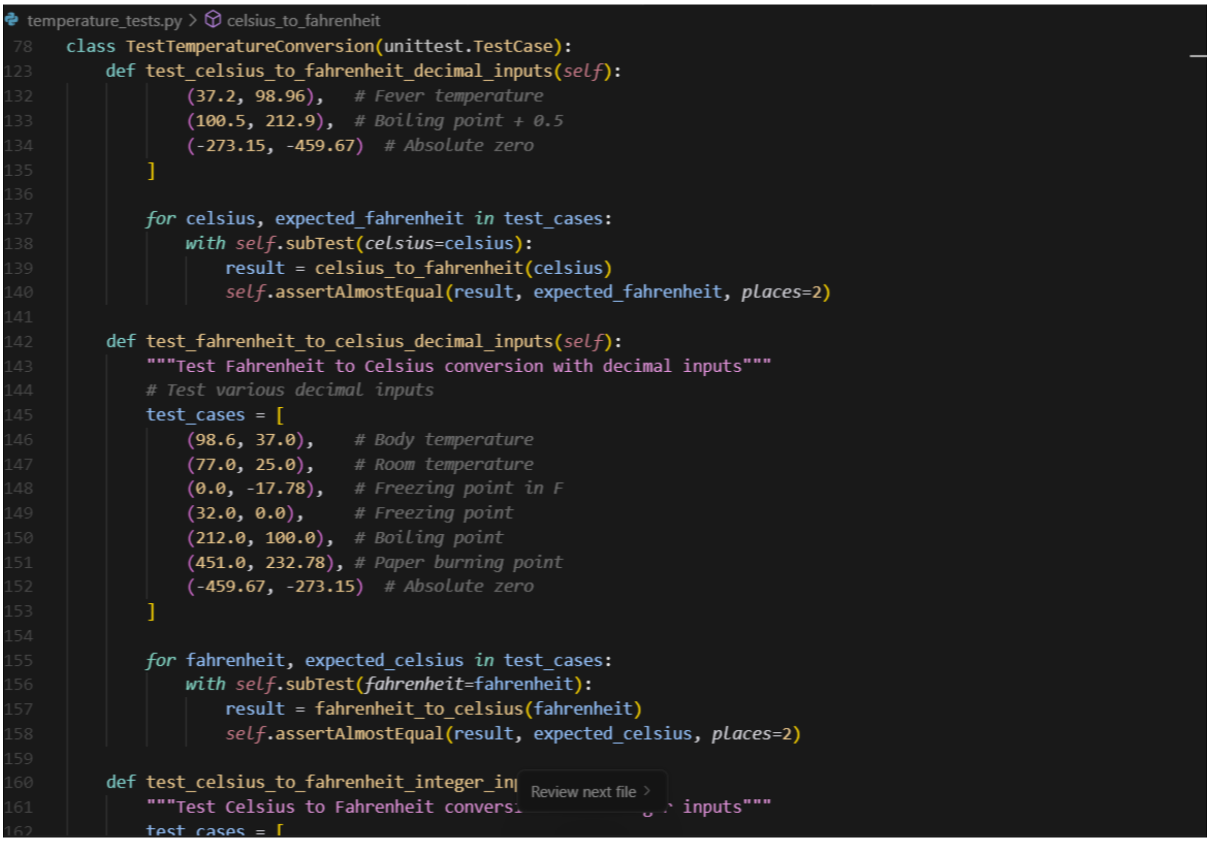
```python
def celsius_to_fahrenheit(c):

    # Convert to float to handle both int and float inputs
    celsius = float(c)

    # Apply the conversion formula
    fahrenheit = (celsius * 9/5) + 32

    return fahrenheit


def fahrenheit_to_celsius(f):
    """
    Convert Fahrenheit to Celsius.

    Args:
        f (float): Temperature in Fahrenheit

    Returns:
        float: Temperature in Celsius

    Formula: C = (F - 32) * 5/9

    Requirements:
        - Validate known pairs: 32°F = 0°C, 212°F = 100°C
        - Handle decimals and invalid inputs like strings or None
    """
    # Handle invalid inputs
    if f is None:
        raise ValueError("Invalid input: None is not a valid temperature")

    # Check for boolean values (True/False are               , int in Python)
    if isinstance(f, bool):
```



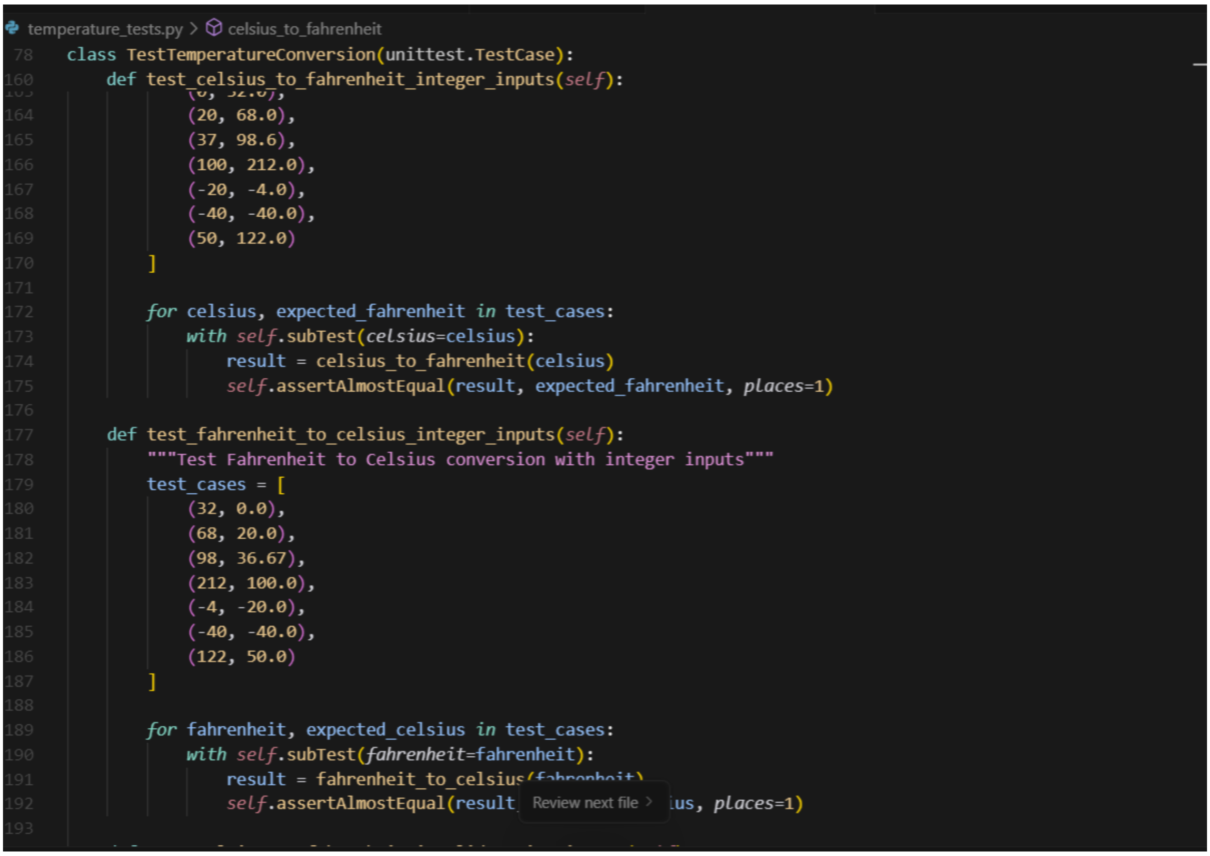
```python
def fahrenheit_to_celsius(f):
    # Check for non-numeric types
    if not isinstance(f, (int, float)):
        raise ValueError(f"Invalid input: {f} is not a number")

    # Convert to float to handle both int and float inputs
    fahrenheit = float(f)

    # Apply the conversion formula
    celsius = (fahrenheit - 32) * 5/9

    return celsius


class TestTemperatureConversion(unittest.TestCase):
    """Test cases for temperature conversion functions"""

    def test_celsius_to_fahrenheit_known_pairs(self):
        """Test known temperature conversion pairs"""
        # 0°C = 32°F
        self.assertAlmostEqual(celsius_to_fahrenheit(0), 32.0, places=5)

        # 100°C = 212°F
        self.assertAlmostEqual(celsius_to_fahrenheit(100), 212.0, places=5)

        # -40°C = -40°F (special case where both scales are equal)
        self.assertAlmostEqual(celsius_to_fahrenheit(-40), -40.0, places=5)

    def test_fahrenheit_to_celsius_known_pairs(self):
        """Test known temperature conversion pairs"""
        # 32°F = 0°C
        self.assertAlmostEqual(fahrenheit_to_               ), places=5)
```



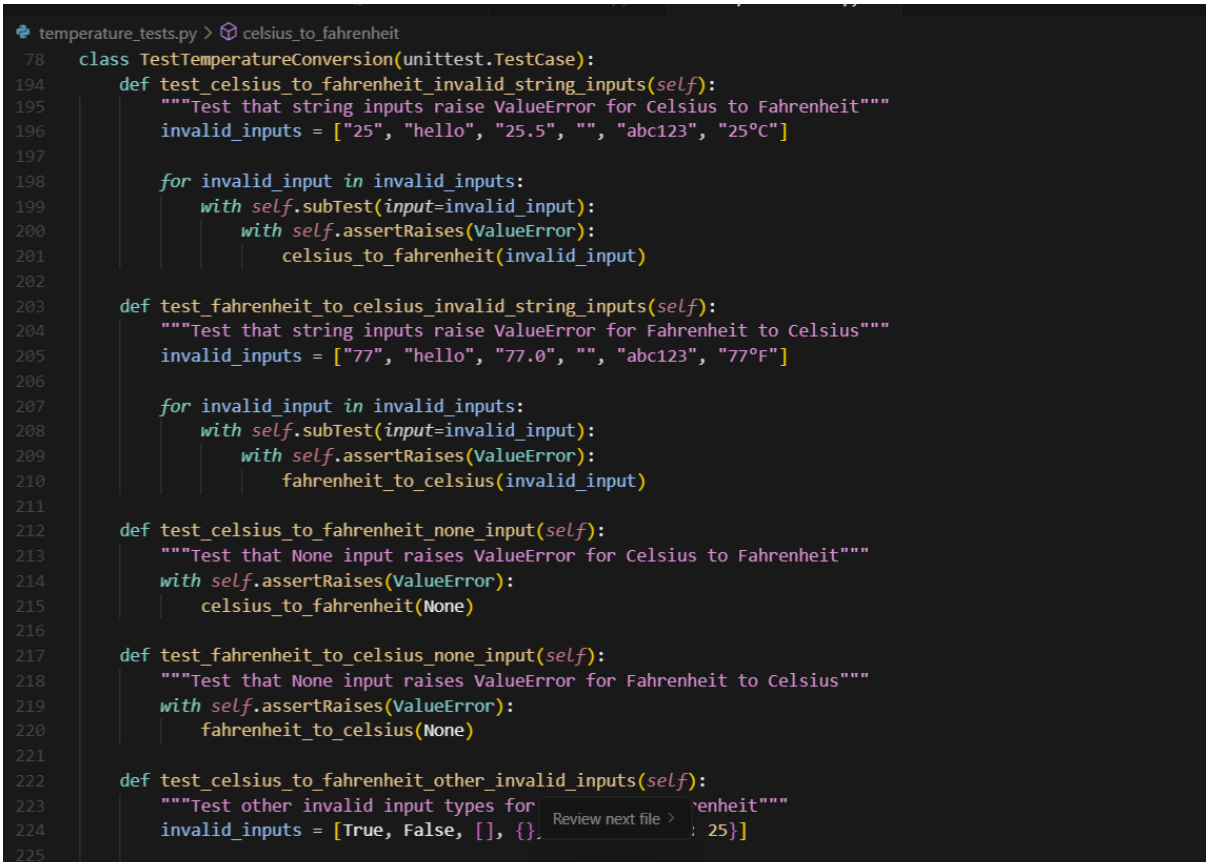
```python
class TestTemperatureConversion(unittest.TestCase):
    def test_fahrenheit_to_celsius_known_pairs(self):
        # -40°F = -40°C (special case where both scales are equal)
        self.assertAlmostEqual(fahrenheit_to_celsius(-40), -40.0, places=5)

    def test_celsius_to_fahrenheit_round_trip(self):
        """Test that converting C->F->C returns original value"""
        test_temperatures = [-50, -20, 0, 20, 37, 50, 100, 150]

        for celsius in test_temperatures:
            fahrenheit = celsius_to_fahrenheit(celsius)
            back_to_celsius = fahrenheit_to_celsius(fahrenheit)
            self.assertAlmostEqual(back_to_celsius, celsius, places=5,
                                   msg=f"Round trip failed for {celsius}°C")

    def test_fahrenheit_to_celsius_round_trip(self):
        """Test that converting F->C->F returns original value"""
        test_temperatures = [-58, -4, 32, 68, 98.6, 122, 212, 302]

        for fahrenheit in test_temperatures:
            celsius = fahrenheit_to_celsius(fahrenheit)
            back_to_fahrenheit = celsius_to_fahrenheit(celsius)
            self.assertAlmostEqual(back_to_fahrenheit, fahrenheit, places=5,
                                   msg=f"Round trip failed for {fahrenheit}°F")

    def test_celsius_to_fahrenheit_decimal_inputs(self):
        """Test Celsius to Fahrenheit conversion with decimal inputs"""
        # Test various decimal inputs
        test_cases = [
            (36.5, 97.7),     # Body temperature
            (25.5, 77.9),     # Room temperatur
            (-18.5, -1.3),    # Freezer temper
            (0.1, 32.18),     # Just above freezing
```



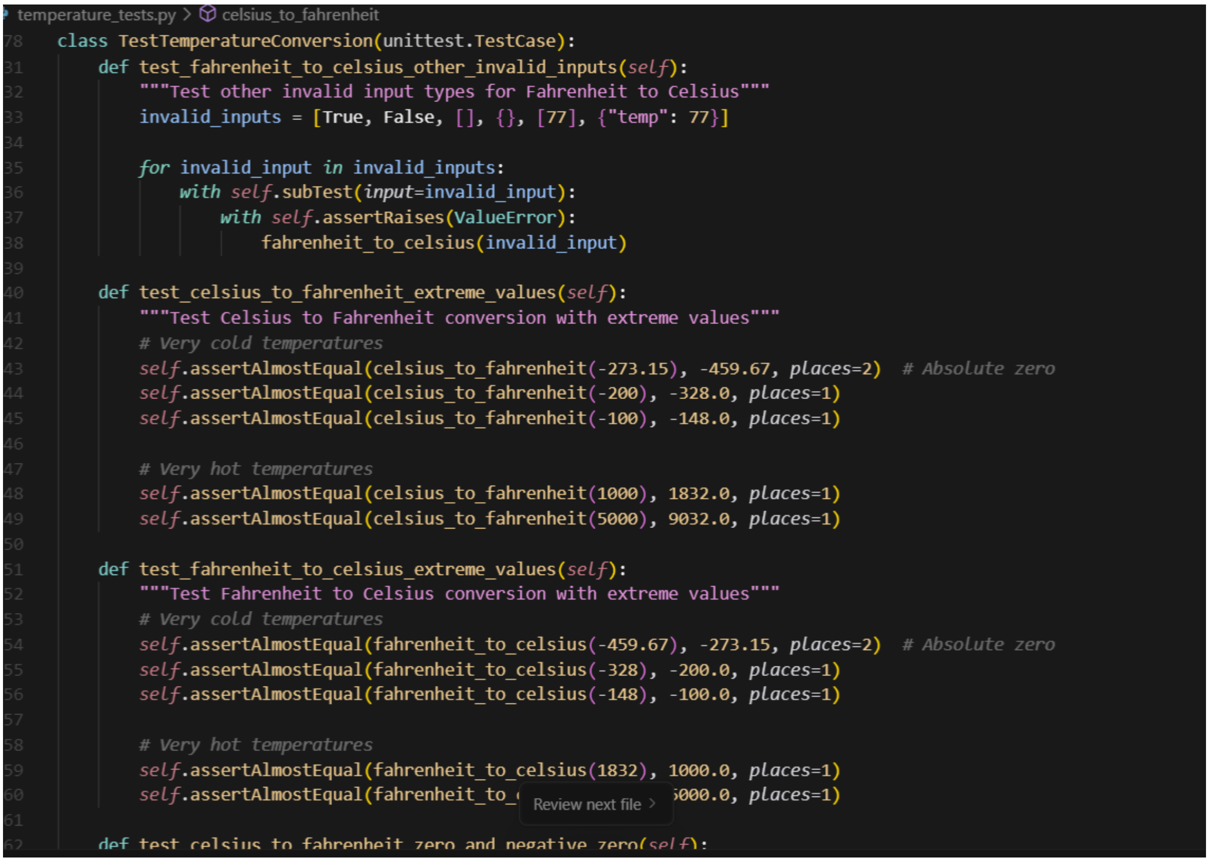
```python
class TestTemperatureConversion(unittest.TestCase):
    def test_celsius_to_fahrenheit_decimal_inputs(self):
            (37.2, 98.96),    # Fever temperature
            (100.5, 212.9),   # Boiling point + 0.5
            (-273.15, -459.67)  # Absolute zero
        ]

        for celsius, expected_fahrenheit in test_cases:
            with self.subTest(celsius=celsius):
                result = celsius_to_fahrenheit(celsius)
                self.assertAlmostEqual(result, expected_fahrenheit, places=2)

    def test_fahrenheit_to_celsius_decimal_inputs(self):
        """Test Fahrenheit to Celsius conversion with decimal inputs"""
        # Test various decimal inputs
        test_cases = [
            (98.6, 37.0),     # Body temperature
            (77.0, 25.0),     # Room temperature
            (0.0, -17.78),    # Freezing point in F
            (32.0, 0.0),      # Freezing point
            (212.0, 100.0),   # Boiling point
            (451.0, 232.78),  # Paper burning point
            (-459.67, -273.15)  # Absolute zero
        ]

        for fahrenheit, expected_celsius in test_cases:
            with self.subTest(fahrenheit=fahrenheit):
                result = fahrenheit_to_celsius(fahrenheit)
                self.assertAlmostEqual(result, expected_celsius, places=2)

    def test_celsius_to_fahrenheit_integer_in
        """Test Celsius to Fahrenheit conversi                inputs"""
        test_cases = [
```



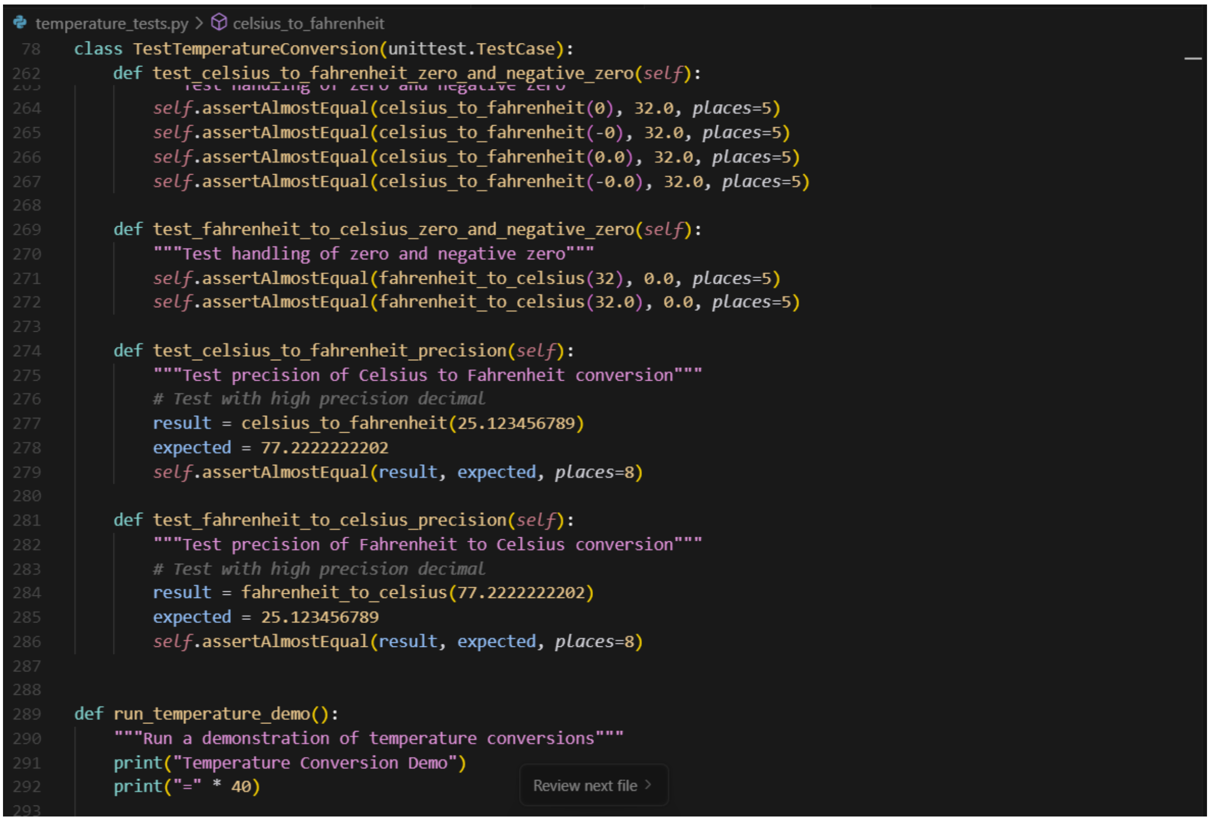
```python
class TestTemperatureConversion(unittest.TestCase):
    def test_celsius_to_fahrenheit_integer_inputs(self):
            (0, 32.0),
            (20, 68.0),
            (37, 98.6),
            (100, 212.0),
            (-20, -4.0),
            (-40, -40.0),
            (50, 122.0)
        ]

        for celsius, expected_fahrenheit in test_cases:
            with self.subTest(celsius=celsius):
                result = celsius_to_fahrenheit(celsius)
                self.assertAlmostEqual(result, expected_fahrenheit, places=1)

    def test_fahrenheit_to_celsius_integer_inputs(self):
        """Test Fahrenheit to Celsius conversion with integer inputs"""
        test_cases = [
            (32, 0.0),
            (68, 20.0),
            (98, 36.67),
            (212, 100.0),
            (-4, -20.0),
            (-40, -40.0),
            (122, 50.0)
        ]

        for fahrenheit, expected_celsius in test_cases:
            with self.subTest(fahrenheit=fahrenheit):
                result = fahrenheit_to_celsius(fahrenheit)
                self.assertAlmostEqual(result                 ius, places=1)
```



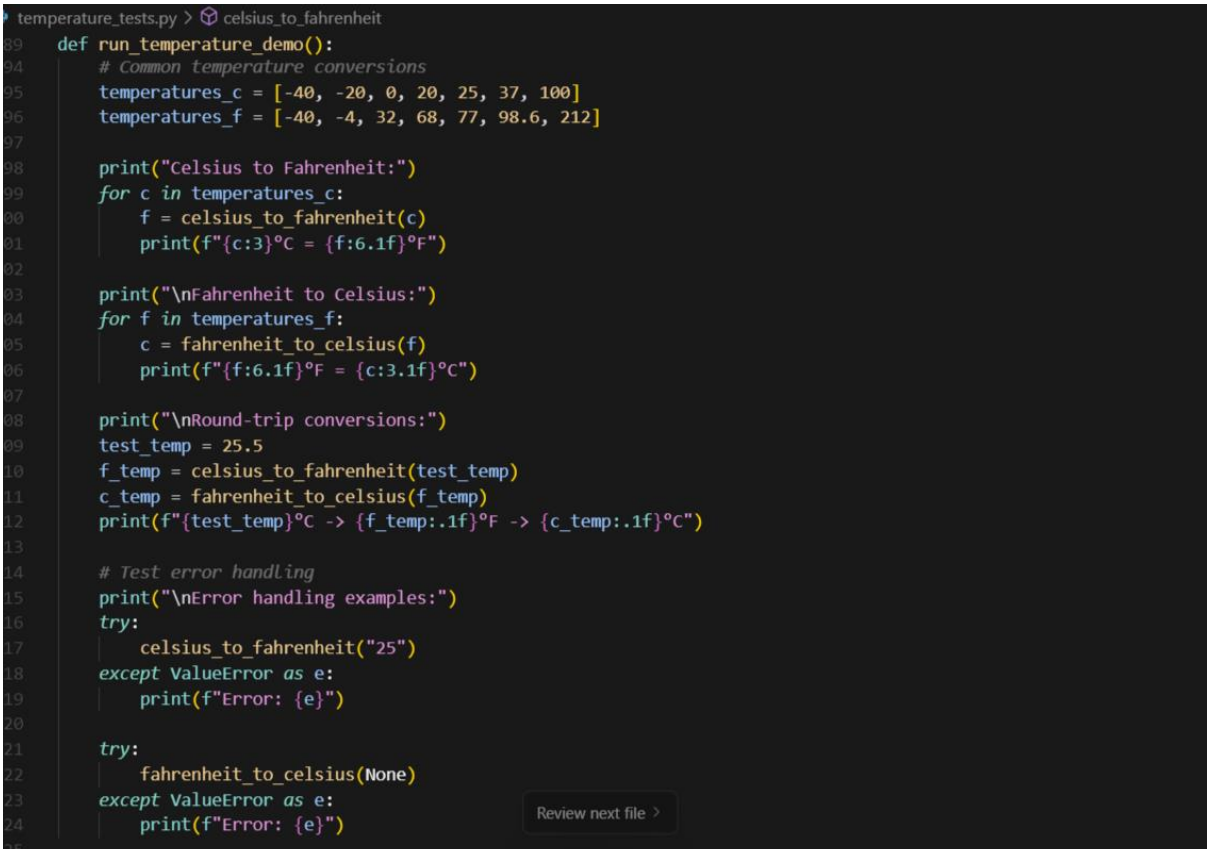
```python
class TestTemperatureConversion(unittest.TestCase):
    def test_celsius_to_fahrenheit_invalid_string_inputs(self):
        """Test that string inputs raise ValueError for Celsius to Fahrenheit"""
        invalid_inputs = ["25", "hello", "25.5", "", "abc123", "25°C"]

        for invalid_input in invalid_inputs:
            with self.subTest(input=invalid_input):
                with self.assertRaises(ValueError):
                    celsius_to_fahrenheit(invalid_input)

    def test_fahrenheit_to_celsius_invalid_string_inputs(self):
        """Test that string inputs raise ValueError for Fahrenheit to Celsius"""
        invalid_inputs = ["77", "hello", "77.0", "", "abc123", "77°F"]

        for invalid_input in invalid_inputs:
            with self.subTest(input=invalid_input):
                with self.assertRaises(ValueError):
                    fahrenheit_to_celsius(invalid_input)

    def test_celsius_to_fahrenheit_none_input(self):
        """Test that None input raises ValueError for Celsius to Fahrenheit"""
        with self.assertRaises(ValueError):
            celsius_to_fahrenheit(None)

    def test_fahrenheit_to_celsius_none_input(self):
        """Test that None input raises ValueError for Fahrenheit to Celsius"""
        with self.assertRaises(ValueError):
            fahrenheit_to_celsius(None)

    def test_celsius_to_fahrenheit_other_invalid_inputs(self):
        """Test other invalid input types for                 enheit"""
        invalid_inputs = [True, False, [], {}                 25}]
```



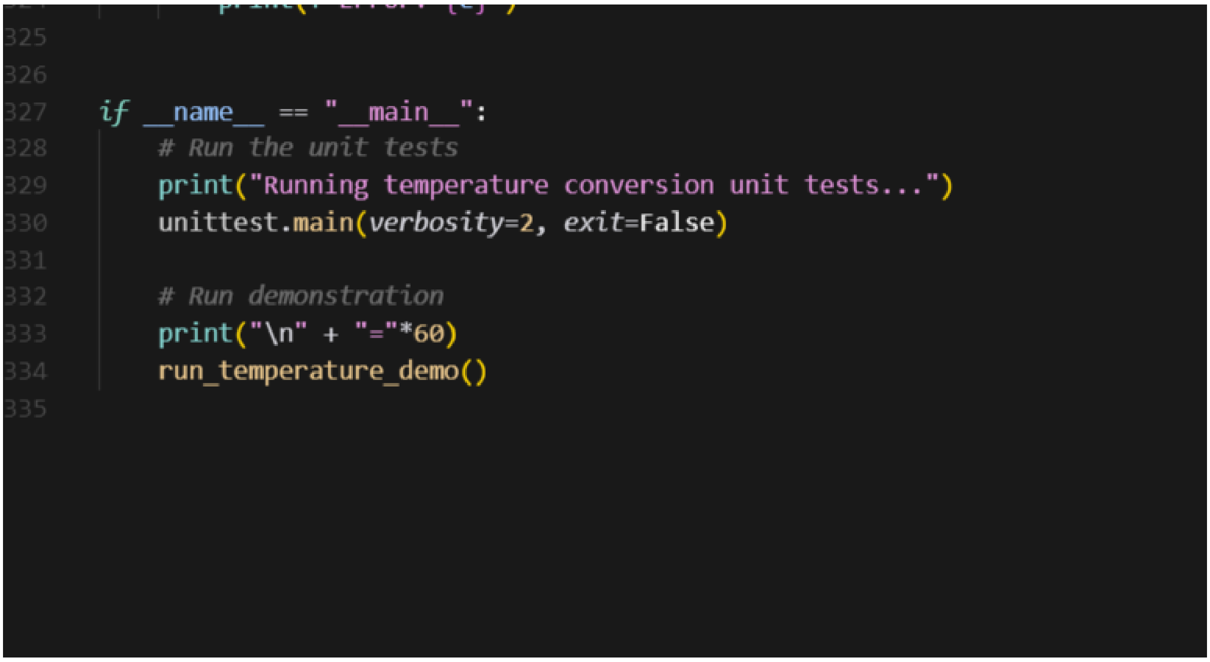
```
class TestTemperatureConversion(unittest.TestCase):
    def test_fahrenheit_to_celsius_other_invalid_inputs(self):
        """Test other invalid input types for Fahrenheit to Celsius"""
        invalid_inputs = [True, False, [], {}, [77], {"temp": 77}]

        for invalid_input in invalid_inputs:
            with self.subTest(input=invalid_input):
                with self.assertRaises(ValueError):
                    fahrenheit_to_celsius(invalid_input)

    def test_celsius_to_fahrenheit_extreme_values(self):
        """Test Celsius to Fahrenheit conversion with extreme values"""
        # Very cold temperatures
        self.assertAlmostEqual(celsius_to_fahrenheit(-273.15), -459.67, places=2)  # Absolute zero
        self.assertAlmostEqual(celsius_to_fahrenheit(-200), -328.0, places=1)
        self.assertAlmostEqual(celsius_to_fahrenheit(-100), -148.0, places=1)

        # Very hot temperatures
        self.assertAlmostEqual(celsius_to_fahrenheit(1000), 1832.0, places=1)
        self.assertAlmostEqual(celsius_to_fahrenheit(5000), 9032.0, places=1)

    def test_fahrenheit_to_celsius_extreme_values(self):
        """Test Fahrenheit to Celsius conversion with extreme values"""
        # Very cold temperatures
        self.assertAlmostEqual(fahrenheit_to_celsius(-459.67), -273.15, places=2)  # Absolute zero
        self.assertAlmostEqual(fahrenheit_to_celsius(-328), -200.0, places=1)
        self.assertAlmostEqual(fahrenheit_to_celsius(-148), -100.0, places=1)

        # Very hot temperatures
        self.assertAlmostEqual(fahrenheit_to_celsius(1832), 1000.0, places=1)
        self.assertAlmostEqual(fahrenheit_to_                  000.0, places=1)

    def test_celsius_to_fahrenheit_zero_and_negative_zero(self):
```



```
class TestTemperatureConversion(unittest.TestCase):
    def test_celsius_to_fahrenheit_zero_and_negative_zero(self):
        Test handling of zero and negative zero
        self.assertAlmostEqual(celsius_to_fahrenheit(0), 32.0, places=5)
        self.assertAlmostEqual(celsius_to_fahrenheit(-0), 32.0, places=5)
        self.assertAlmostEqual(celsius_to_fahrenheit(0.0), 32.0, places=5)
        self.assertAlmostEqual(celsius_to_fahrenheit(-0.0), 32.0, places=5)

    def test_fahrenheit_to_celsius_zero_and_negative_zero(self):
        """Test handling of zero and negative zero"""
        self.assertAlmostEqual(fahrenheit_to_celsius(32), 0.0, places=5)
        self.assertAlmostEqual(fahrenheit_to_celsius(32.0), 0.0, places=5)

    def test_celsius_to_fahrenheit_precision(self):
        """Test precision of Celsius to Fahrenheit conversion"""
        # Test with high precision decimal
        result = celsius_to_fahrenheit(25.123456789)
        expected = 77.2222222202
        self.assertAlmostEqual(result, expected, places=8)

    def test_fahrenheit_to_celsius_precision(self):
        """Test precision of Fahrenheit to Celsius conversion"""
        # Test with high precision decimal
        result = fahrenheit_to_celsius(77.2222222202)
        expected = 25.123456789
        self.assertAlmostEqual(result, expected, places=8)


def run_temperature_demo():
    """Run a demonstration of temperature conversions"""
    print("Temperature Conversion Demo")
    print("=" * 40)
```

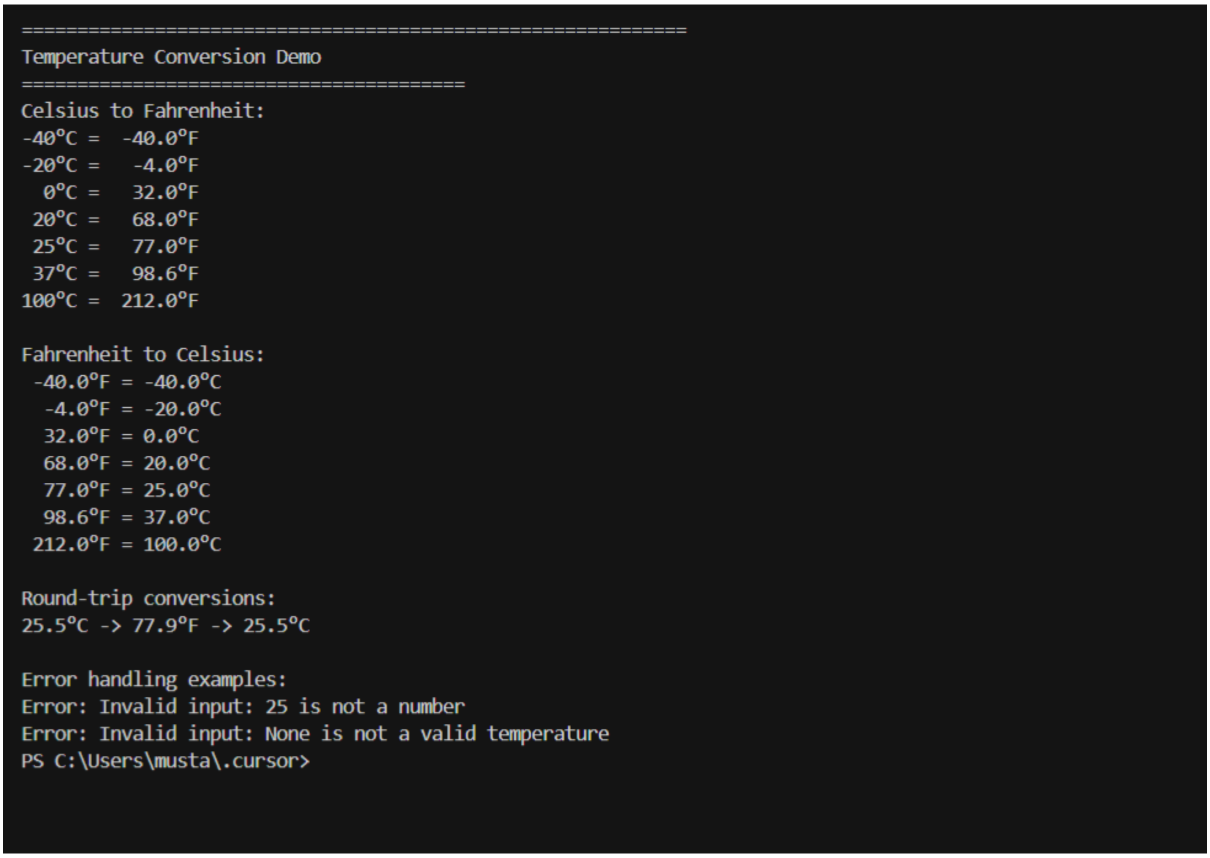


```python
def run_temperature_demo():
    # Common temperature conversions
    temperatures_c = [-40, -20, 0, 20, 25, 37, 100]
    temperatures_f = [-40, -4, 32, 68, 77, 98.6, 212]

    print("Celsius to Fahrenheit:")
    for c in temperatures_c:
        f = celsius_to_fahrenheit(c)
        print(f"{c:3}°C = {f:6.1f}°F")

    print("\nFahrenheit to Celsius:")
    for f in temperatures_f:
        c = fahrenheit_to_celsius(f)
        print(f"{f:6.1f}°F = {c:3.1f}°C")

    print("\nRound-trip conversions:")
    test_temp = 25.5
    f_temp = celsius_to_fahrenheit(test_temp)
    c_temp = fahrenheit_to_celsius(f_temp)
    print(f"{test_temp}°C -> {f_temp:.1f}°F -> {c_temp:.1f}°C")

    # Test error handling
    print("\nError handling examples:")
    try:
        celsius_to_fahrenheit("25")
    except ValueError as e:
        print(f"Error: {e}")

    try:
        fahrenheit_to_celsius(None)
    except ValueError as e:
        print(f"Error: {e}")
```



```python


if __name__ == "__main__":
    # Run the unit tests
    print("Running temperature conversion unit tests...")
    unittest.main(verbosity=2, exit=False)

    # Run demonstration
    print("\n" + "="*60)
    run_temperature_demo()

```

Output:



```
============================================================
Temperature Conversion Demo
========================================
Celsius to Fahrenheit:
-40°C =  -40.0°F
-20°C =   -4.0°F
  0°C =   32.0°F
 20°C =   68.0°F
 25°C =   77.0°F
 37°C =   98.6°F
100°C =  212.0°F

Fahrenheit to Celsius:
 -40.0°F = -40.0°C
  -4.0°F = -20.0°C
  32.0°F = 0.0°C
  68.0°F = 20.0°C
  77.0°F = 25.0°C
  98.6°F = 37.0°C
 212.0°F = 100.0°C

Round-trip conversions:
25.5°C -> 77.9°F -> 25.5°C

Error handling examples:
Error: Invalid input: 25 is not a number
Error: Invalid input: None is not a valid temperature
PS C:\Users\musta\.cursor>
```

Observation:

The output displays the Temperature Conversion Demo script. This program is a simple utility that demonstrates temperature conversions between Celsius and Fahrenheit

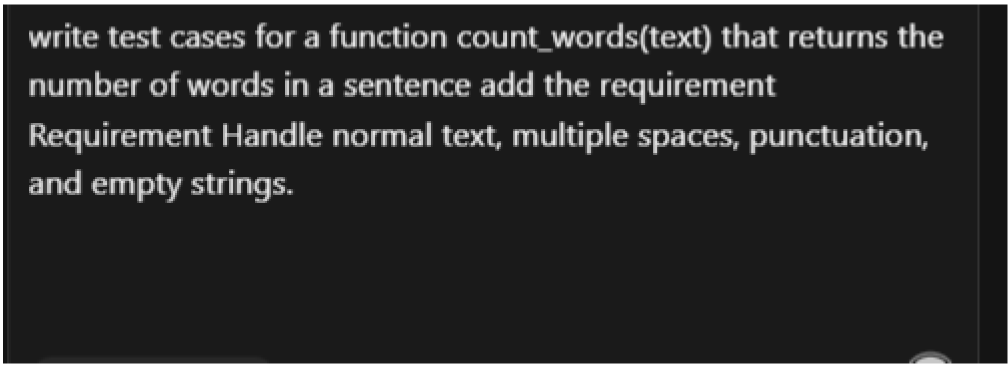
Task Description#3

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.

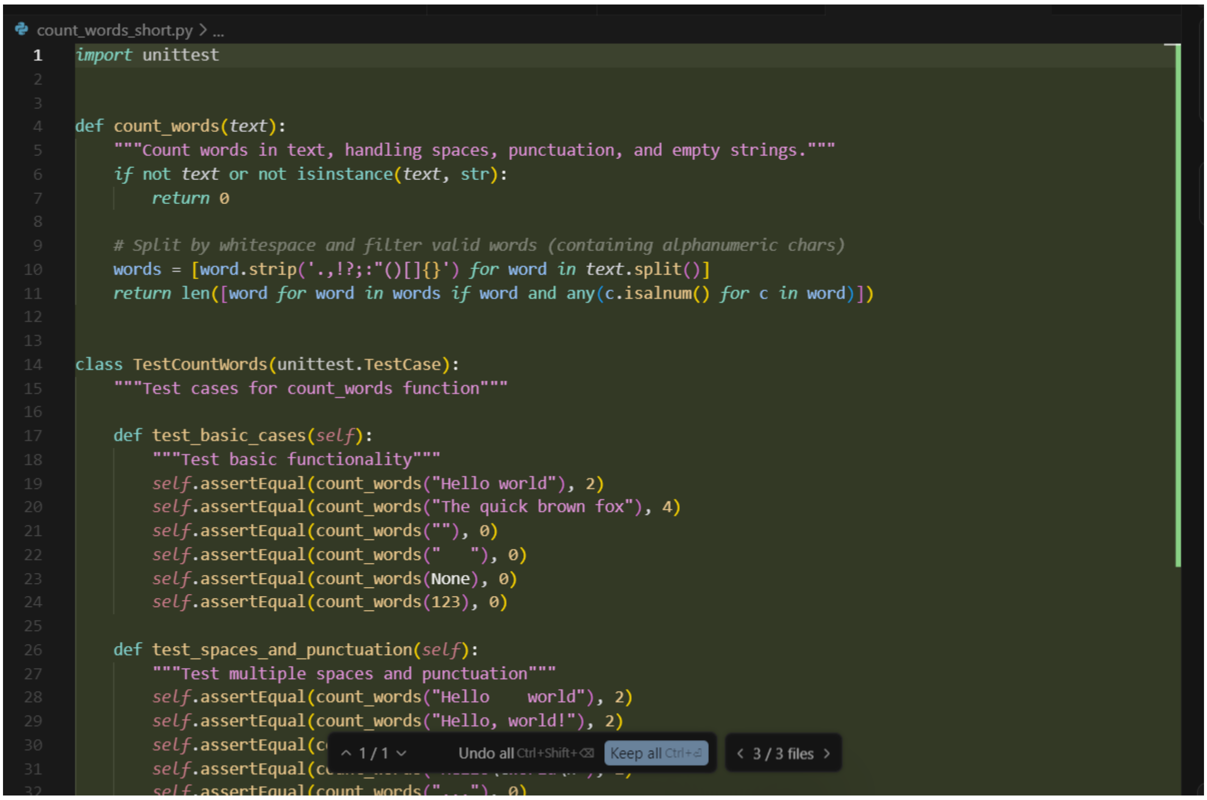
Given prompt to



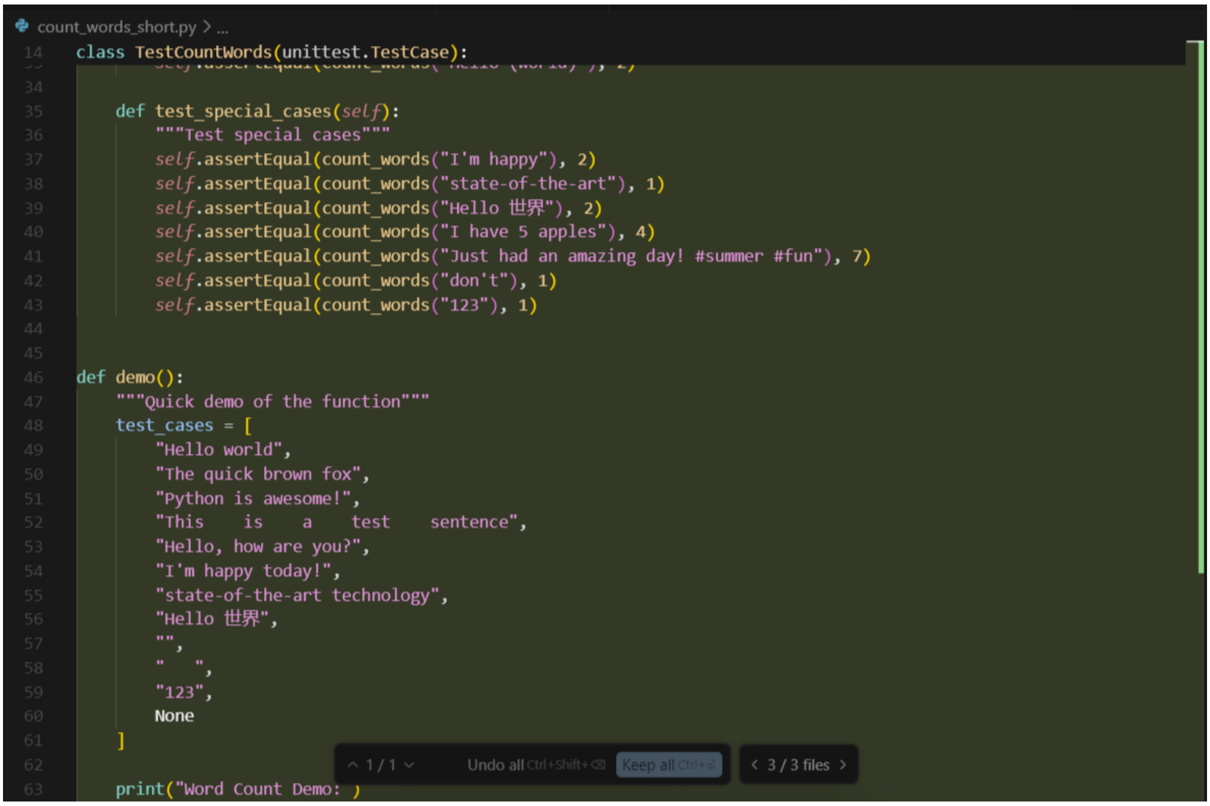
write test cases for a function count_words(text) that returns the number of words in a sentence add the requirement Requirement Handle normal text, multiple spaces, punctuation, and empty strings.

AI:

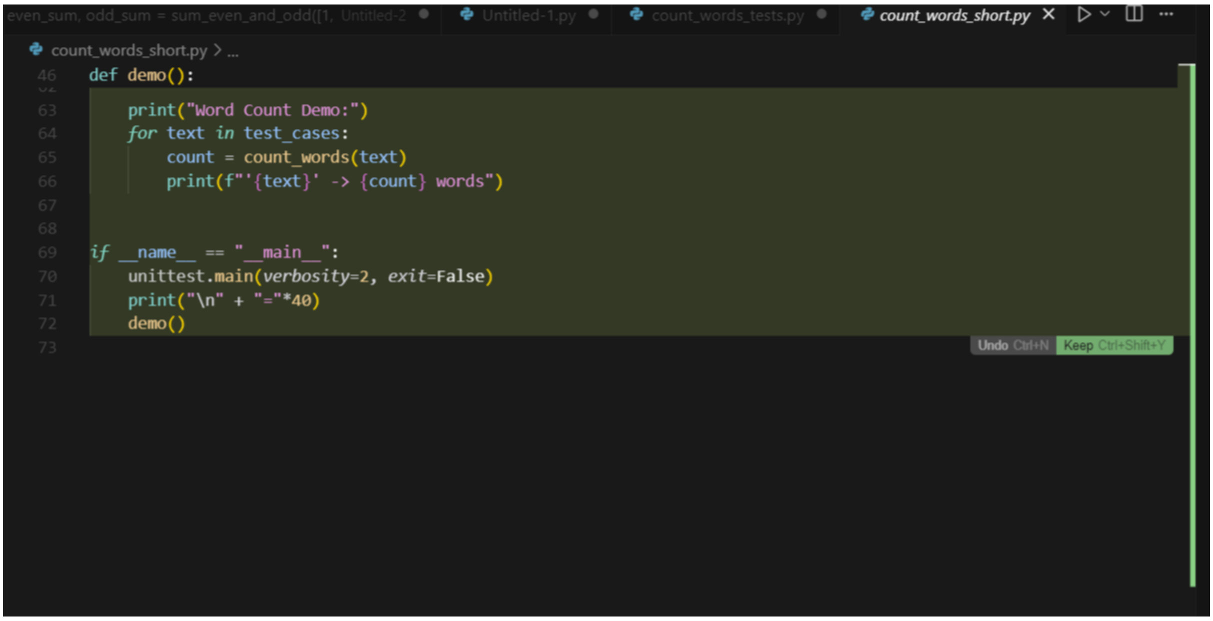
Given code by AI:



```python
import unittest


def count_words(text):
    """Count words in text, handling spaces, punctuation, and empty strings."""
    if not text or not isinstance(text, str):
        return 0

    # Split by whitespace and filter valid words (containing alphanumeric chars)
    words = [word.strip('.,!?;:"()[]{}') for word in text.split()]
    return len([word for word in words if word and any(c.isalnum() for c in word)])


class TestCountWords(unittest.TestCase):
    """Test cases for count_words function"""

    def test_basic_cases(self):
        """Test basic functionality"""
        self.assertEqual(count_words("Hello world"), 2)
        self.assertEqual(count_words("The quick brown fox"), 4)
        self.assertEqual(count_words(""), 0)
        self.assertEqual(count_words("   "), 0)
        self.assertEqual(count_words(None), 0)
        self.assertEqual(count_words(123), 0)

    def test_spaces_and_punctuation(self):
        """Test multiple spaces and punctuation"""
        self.assertEqual(count_words("Hello    world"), 2)
        self.assertEqual(count_words("Hello, world!"), 2)
        self.assertEqual(c
        self.assertEqual(c
        self.assertEqual(count_words("..."), 0)
```

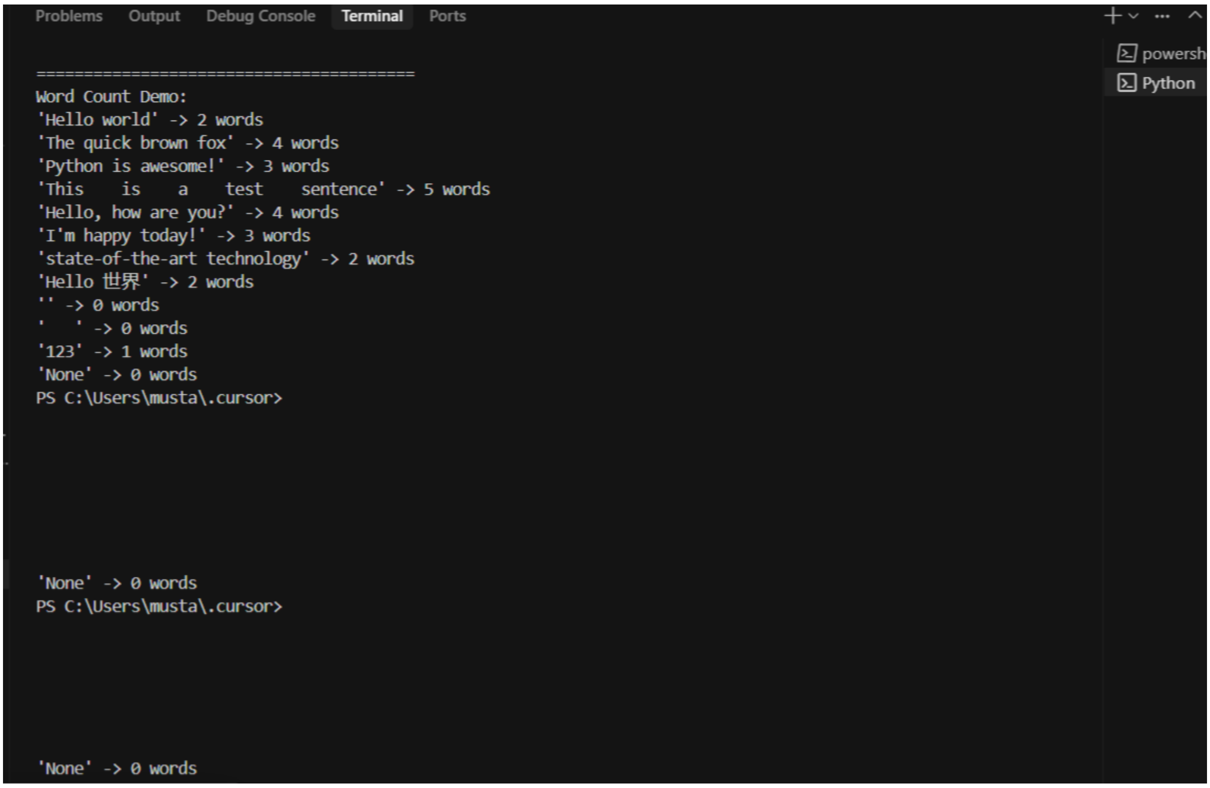


```
class TestCountWords(unittest.TestCase):

    def test_special_cases(self):
        """Test special cases"""
        self.assertEqual(count_words("I'm happy"), 2)
        self.assertEqual(count_words("state-of-the-art"), 1)
        self.assertEqual(count_words("Hello 世界"), 2)
        self.assertEqual(count_words("I have 5 apples"), 4)
        self.assertEqual(count_words("Just had an amazing day! #summer #fun"), 7)
        self.assertEqual(count_words("don't"), 1)
        self.assertEqual(count_words("123"), 1)


def demo():
    """Quick demo of the function"""
    test_cases = [
        "Hello world",
        "The quick brown fox",
        "Python is awesome!",
        "This    is    a    test    sentence",
        "Hello, how are you?",
        "I'm happy today!",
        "state-of-the-art technology",
        "Hello 世界",
        "",
        "   ",
        "123",
        None
    ]

    print("Word Count Demo:")
```

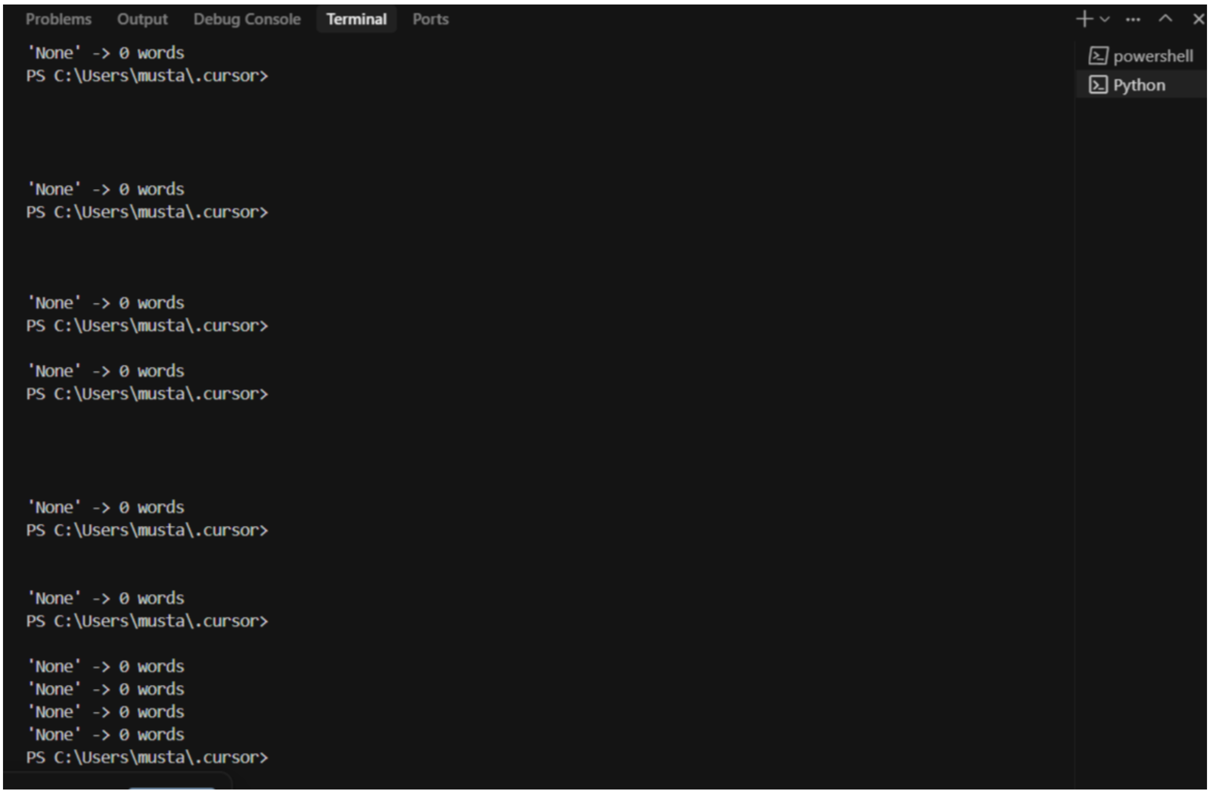


```
def demo():
    print("Word Count Demo:")
    for text in test_cases:
        count = count_words(text)
        print(f"'{text}' -> {count} words")


if __name__ == "__main__":
    unittest.main(verbosity=2, exit=False)
    print("\n" + "="*40)
    demo()
```

Output:



```
========================================
Word Count Demo:
'Hello world' -> 2 words
'The quick brown fox' -> 4 words
'Python is awesome!' -> 3 words
'This    is    a    test    sentence' -> 5 words
'Hello, how are you?' -> 4 words
'I'm happy today!' -> 3 words
'state-of-the-art technology' -> 2 words
'Hello 世界' -> 2 words
'' -> 0 words
'   ' -> 0 words
'123' -> 1 words
'None' -> 0 words
PS C:\Users\musta\.cursor>






 'None' -> 0 words
PS C:\Users\musta\.cursor>






 'None' -> 0 words
```



```
'None' -> 0 words
PS C:\Users\musta\.cursor>




'None' -> 0 words
PS C:\Users\musta\.cursor>



'None' -> 0 words
PS C:\Users\musta\.cursor>

'None' -> 0 words
PS C:\Users\musta\.cursor>




'None' -> 0 words
PS C:\Users\musta\.cursor>


'None' -> 0 words
PS C:\Users\musta\.cursor>

'None' -> 0 words
'None' -> 0 words
'None' -> 0 words
'None' -> 0 words
PS C:\Users\musta\.cursor>
```

Observation:

The output of a script that appears to be counting words from an input. The output shows multiple lines, each with the same result: 'None' -> 0 words

Task Description#4

- Generate test cases for a BankAccount class with:

Methods:

deposit(amount)

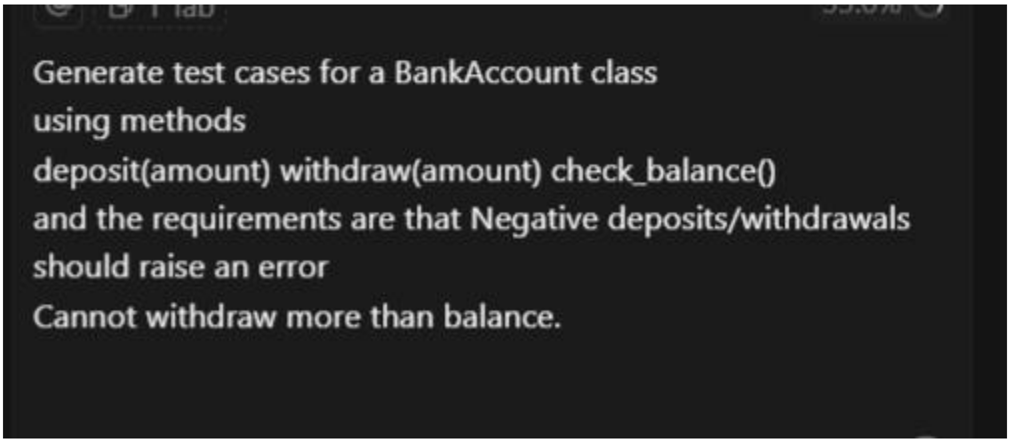
withdraw(amount)

check_balance()

Requirements:

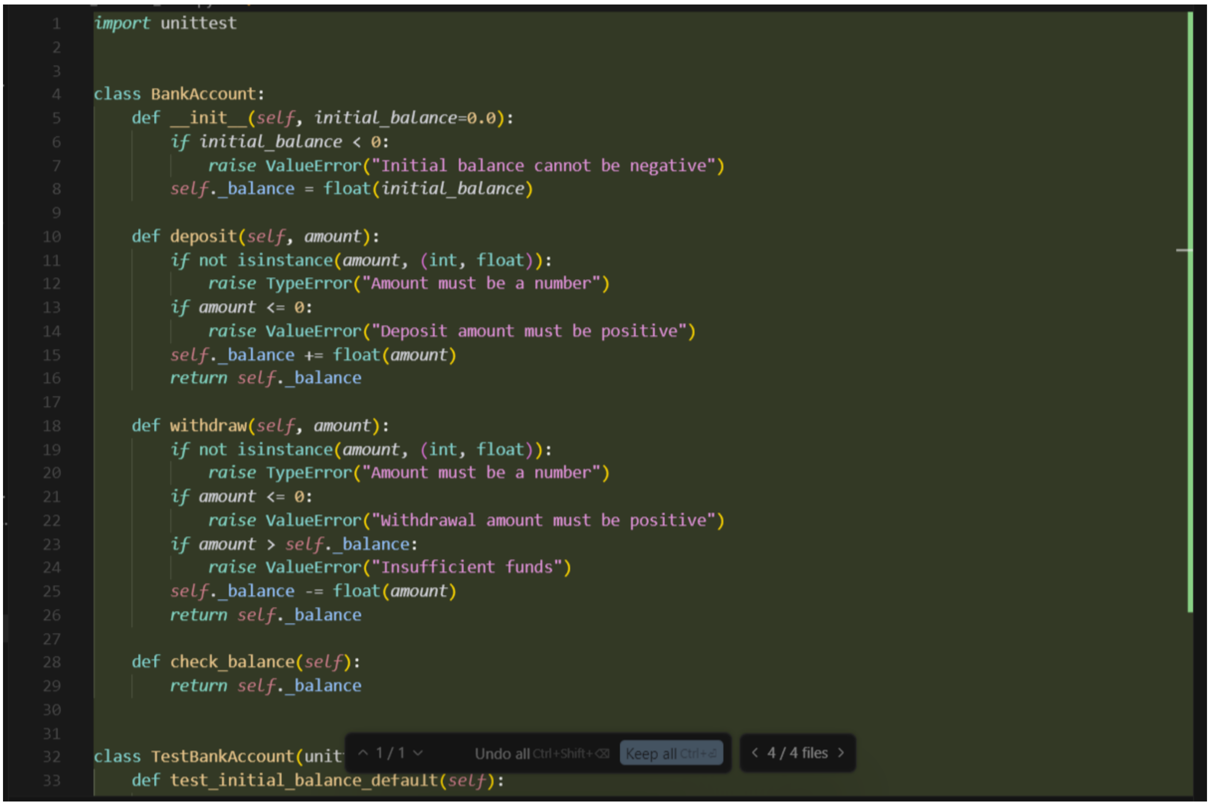
- Negative deposits/withdrawals should raise an error.
- Cannot withdraw more than balance

Given promt to the AI:

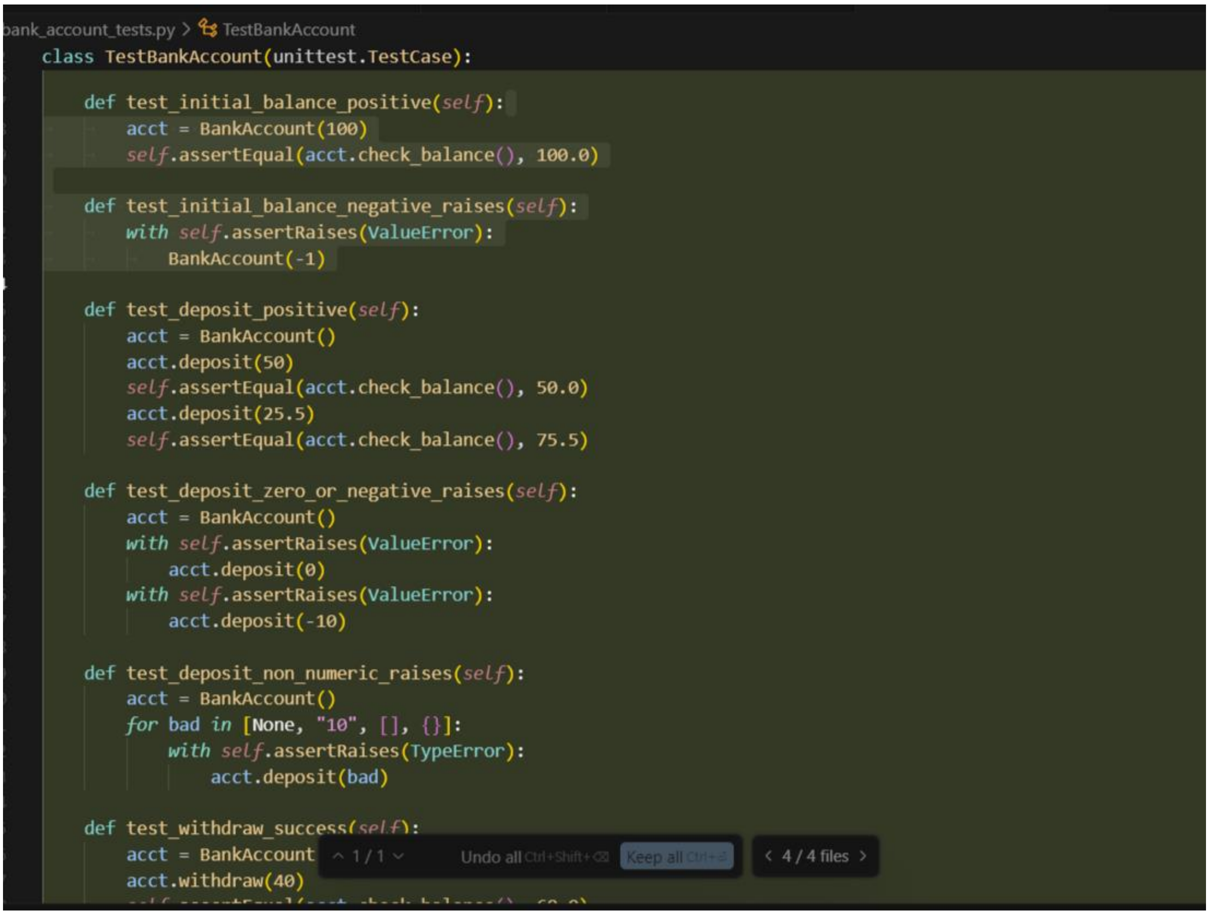


Generate test cases for a BankAccount class
using methods
deposit(amount) withdraw(amount) check_balance()
and the requirements are that Negative deposits/withdrawals
should raise an error
Cannot withdraw more than balance.

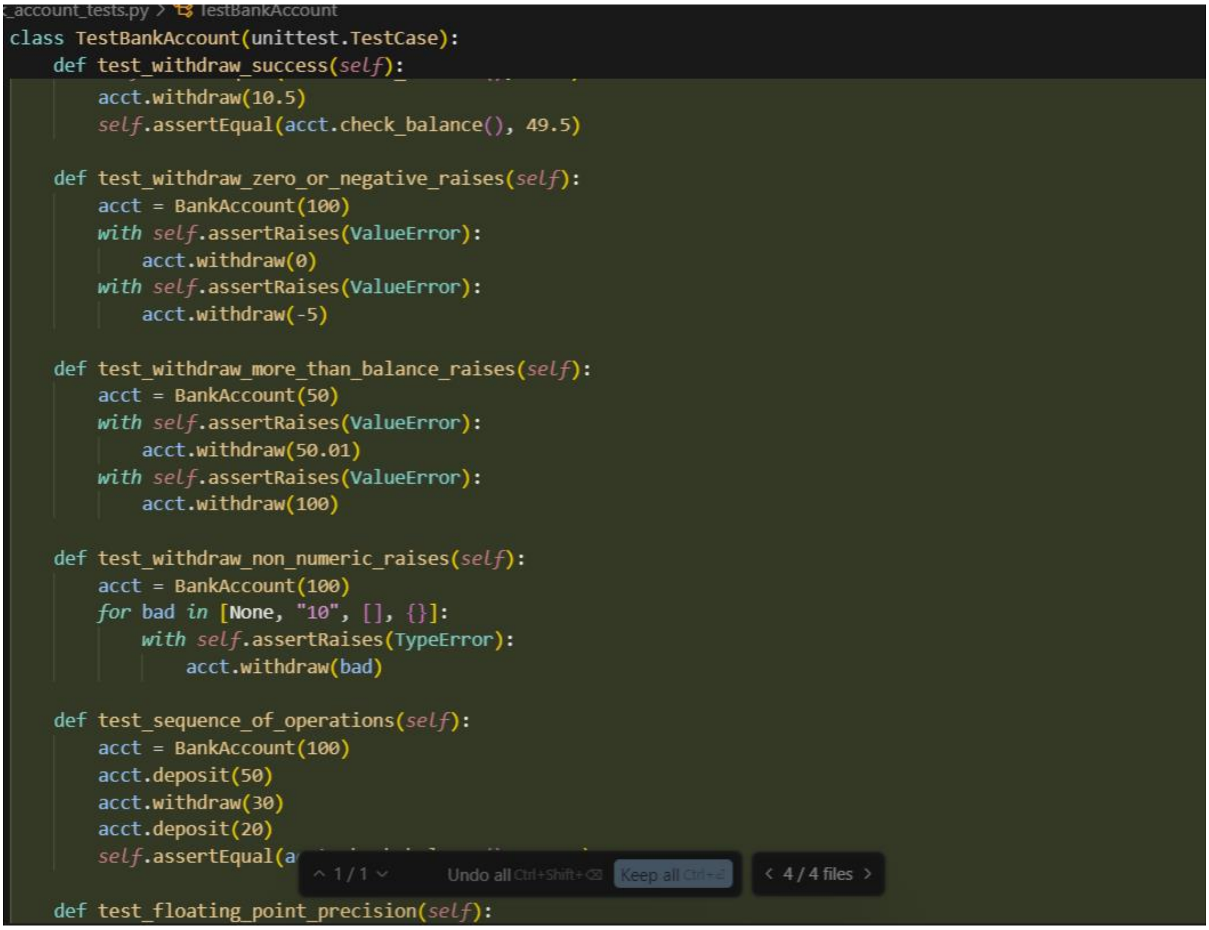
Given program by the AI:



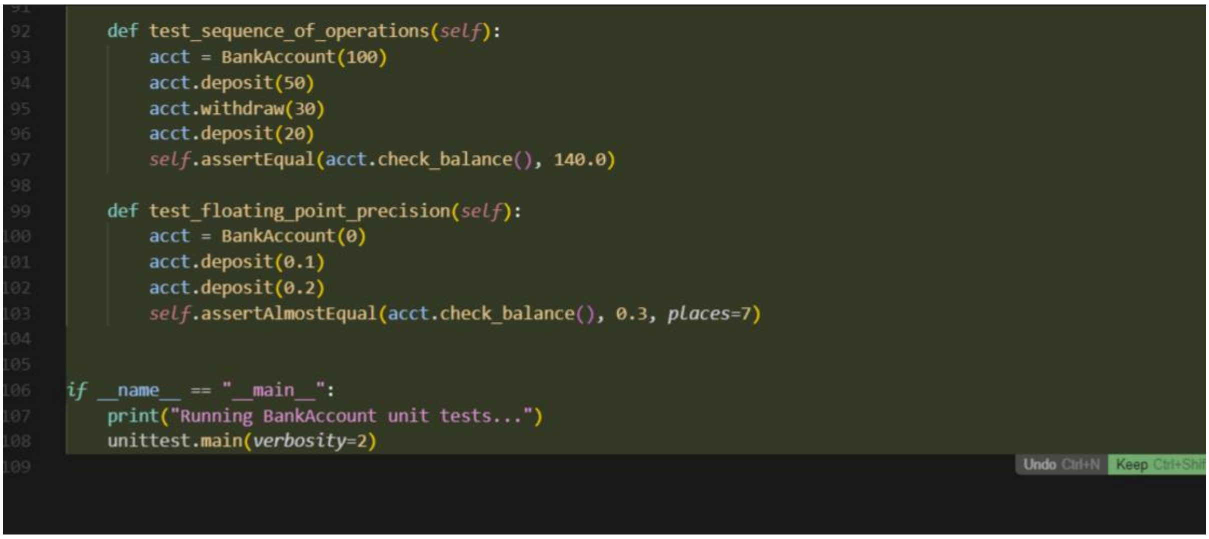
```
import unittest


class BankAccount:
    def __init__(self, initial_balance=0.0):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative")
        self._balance = float(initial_balance)

    def deposit(self, amount):
        if not isinstance(amount, (int, float)):
            raise TypeError("Amount must be a number")
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self._balance += float(amount)
        return self._balance

    def withdraw(self, amount):
        if not isinstance(amount, (int, float)):
            raise TypeError("Amount must be a number")
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self._balance:
            raise ValueError("Insufficient funds")
        self._balance -= float(amount)
        return self._balance

    def check_balance(self):
        return self._balance


class TestBankAccount(unit
    def test_initial_balance_default(self):
```



```python
class TestBankAccount(unittest.TestCase):

    def test_initial_balance_positive(self):
        acct = BankAccount(100)
        self.assertEqual(acct.check_balance(), 100.0)

    def test_initial_balance_negative_raises(self):
        with self.assertRaises(ValueError):
            BankAccount(-1)

    def test_deposit_positive(self):
        acct = BankAccount()
        acct.deposit(50)
        self.assertEqual(acct.check_balance(), 50.0)
        acct.deposit(25.5)
        self.assertEqual(acct.check_balance(), 75.5)

    def test_deposit_zero_or_negative_raises(self):
        acct = BankAccount()
        with self.assertRaises(ValueError):
            acct.deposit(0)
        with self.assertRaises(ValueError):
            acct.deposit(-10)

    def test_deposit_non_numeric_raises(self):
        acct = BankAccount()
        for bad in [None, "10", [], {}]:
            with self.assertRaises(TypeError):
                acct.deposit(bad)

    def test_withdraw_success(self):
        acct = BankAccount
        acct.withdraw(40)
```

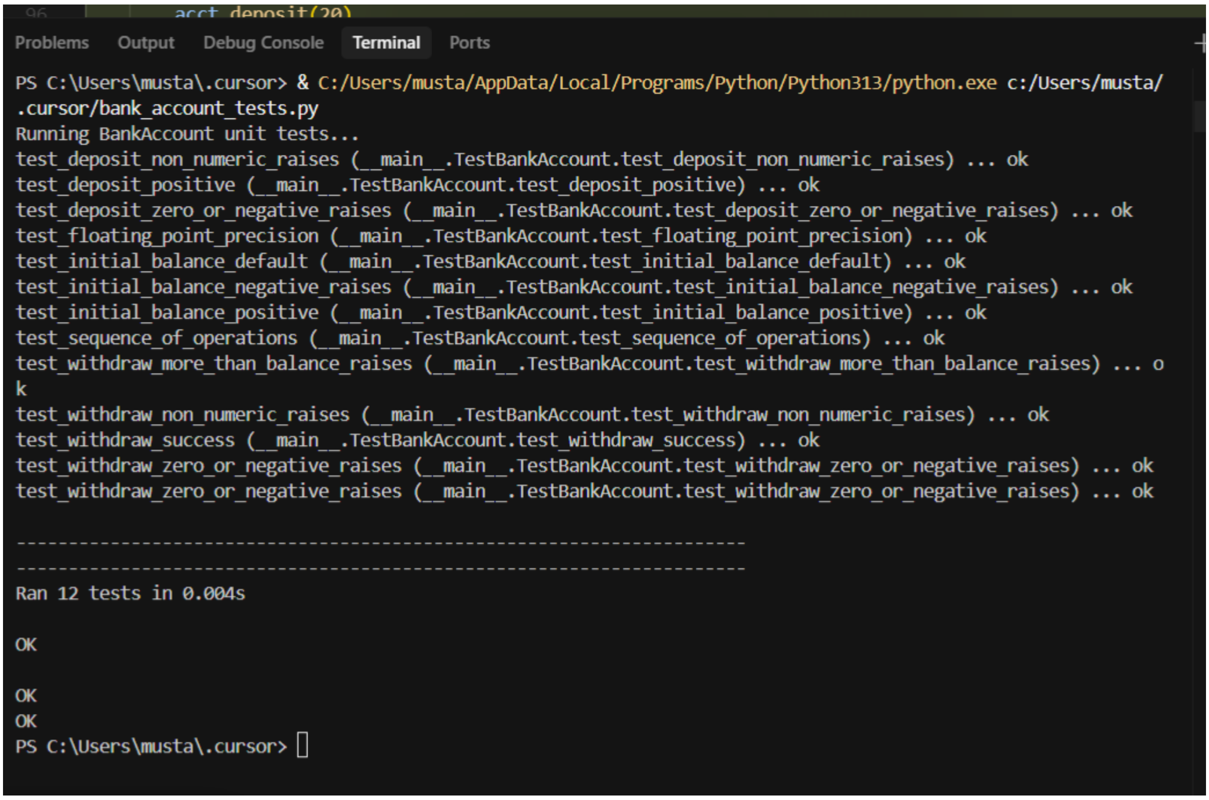


```python
class TestBankAccount(unittest.TestCase):
    def test_withdraw_success(self):
        acct.withdraw(10.5)
        self.assertEqual(acct.check_balance(), 49.5)

    def test_withdraw_zero_or_negative_raises(self):
        acct = BankAccount(100)
        with self.assertRaises(ValueError):
            acct.withdraw(0)
        with self.assertRaises(ValueError):
            acct.withdraw(-5)

    def test_withdraw_more_than_balance_raises(self):
        acct = BankAccount(50)
        with self.assertRaises(ValueError):
            acct.withdraw(50.01)
        with self.assertRaises(ValueError):
            acct.withdraw(100)

    def test_withdraw_non_numeric_raises(self):
        acct = BankAccount(100)
        for bad in [None, "10", [], {}]:
            with self.assertRaises(TypeError):
                acct.withdraw(bad)

    def test_sequence_of_operations(self):
        acct = BankAccount(100)
        acct.deposit(50)
        acct.withdraw(30)
        acct.deposit(20)
        self.assertEqual(a

    def test_floating_point_precision(self):
```



```python
    def test_sequence_of_operations(self):
        acct = BankAccount(100)
        acct.deposit(50)
        acct.withdraw(30)
        acct.deposit(20)
        self.assertEqual(acct.check_balance(), 140.0)

    def test_floating_point_precision(self):
        acct = BankAccount(0)
        acct.deposit(0.1)
        acct.deposit(0.2)
        self.assertAlmostEqual(acct.check_balance(), 0.3, places=7)


if __name__ == "__main__":
    print("Running BankAccount unit tests...")
    unittest.main(verbosity=2)
```

Output:



```
PS C:\Users\musta\.cursor> & C:/Users/musta/AppData/Local/Programs/Python/Python313/python.exe c:/Users/musta/
.cursor/bank_account_tests.py
Running BankAccount unit tests...
test_deposit_non_numeric_raises (__main__.TestBankAccount.test_deposit_non_numeric_raises) ... ok
test_deposit_positive (__main__.TestBankAccount.test_deposit_positive) ... ok
test_deposit_zero_or_negative_raises (__main__.TestBankAccount.test_deposit_zero_or_negative_raises) ... ok
test_floating_point_precision (__main__.TestBankAccount.test_floating_point_precision) ... ok
test_initial_balance_default (__main__.TestBankAccount.test_initial_balance_default) ... ok
test_initial_balance_negative_raises (__main__.TestBankAccount.test_initial_balance_negative_raises) ... ok
test_initial_balance_positive (__main__.TestBankAccount.test_initial_balance_positive) ... ok
test_sequence_of_operations (__main__.TestBankAccount.test_sequence_of_operations) ... ok
test_withdraw_more_than_balance_raises (__main__.TestBankAccount.test_withdraw_more_than_balance_raises) ... o
k
test_withdraw_non_numeric_raises (__main__.TestBankAccount.test_withdraw_non_numeric_raises) ... ok
test_withdraw_success (__main__.TestBankAccount.test_withdraw_success) ... ok
test_withdraw_zero_or_negative_raises (__main__.TestBankAccount.test_withdraw_zero_or_negative_raises) ... ok
test_withdraw_zero_or_negative_raises (__main__.TestBankAccount.test_withdraw_zero_or_negative_raises) ... ok

----------------------------------------------------------------------
----------------------------------------------------------------------
Ran 12 tests in 0.004s

OK

OK
OK
PS C:\Users\musta\.cursor> 
```

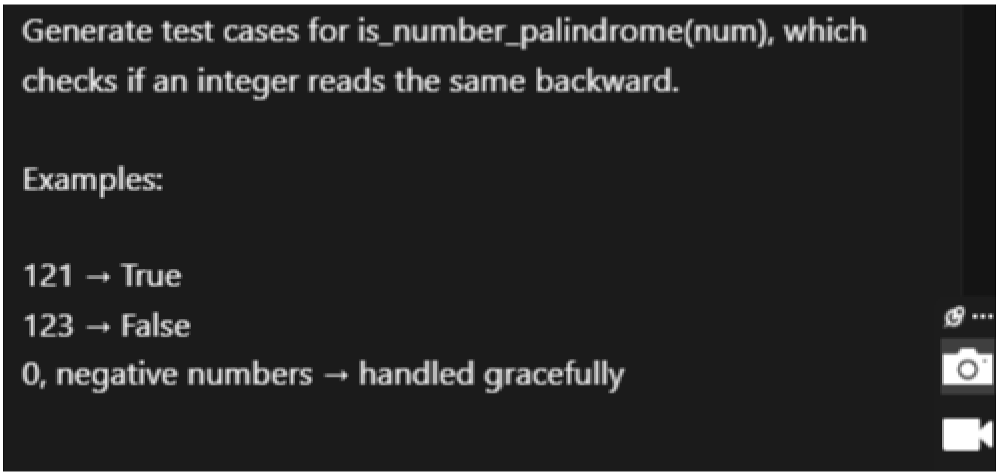
Observation:

All 12 unit tests for the BankAccount class executed successfully with OK status. The results confirm that deposit, withdrawal, balance validation, and precision features are working correctly without errors.

Task Description#5

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

Given prompt to AI:



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

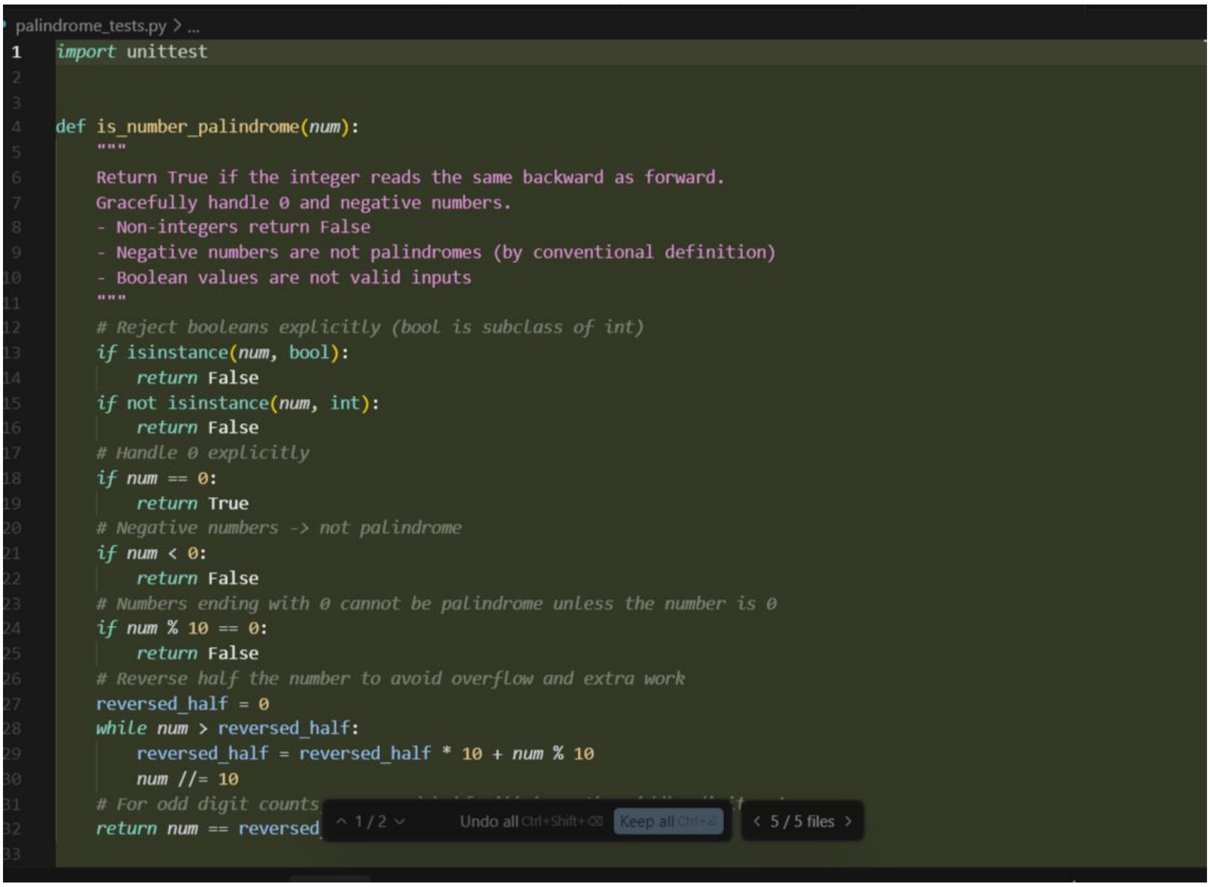
Examples:

121 → True

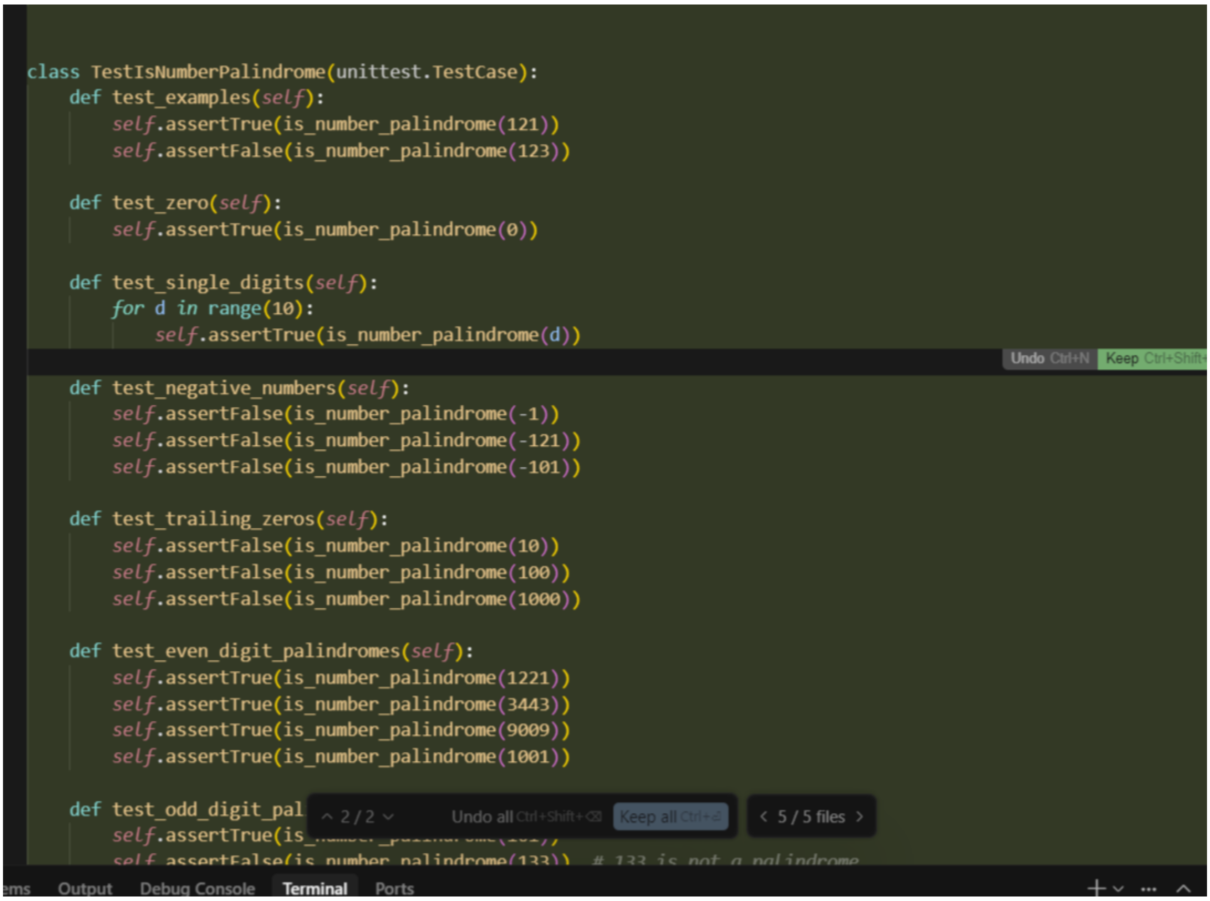
123 → False

0, negative numbers → handled gracefully

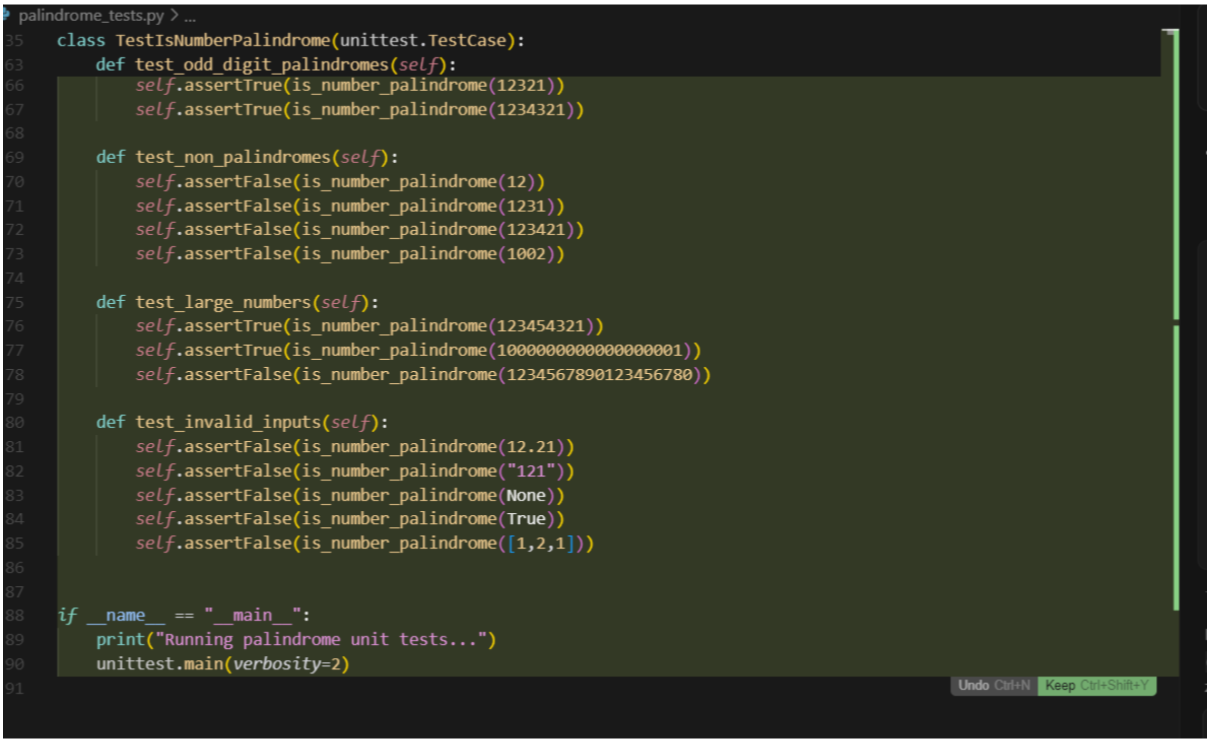
Given program by the AI:



```python
import unittest


def is_number_palindrome(num):
    """
    Return True if the integer reads the same backward as forward.
    Gracefully handle 0 and negative numbers.
    - Non-integers return False
    - Negative numbers are not palindromes (by conventional definition)
    - Boolean values are not valid inputs
    """
    # Reject booleans explicitly (bool is subclass of int)
    if isinstance(num, bool):
        return False
    if not isinstance(num, int):
        return False
    # Handle 0 explicitly
    if num == 0:
        return True
    # Negative numbers -> not palindrome
    if num < 0:
        return False
    # Numbers ending with 0 cannot be palindrome unless the number is 0
    if num % 10 == 0:
        return False
    # Reverse half the number to avoid overflow and extra work
    reversed_half = 0
    while num > reversed_half:
        reversed_half = reversed_half * 10 + num % 10
        num //= 10
    # For odd digit counts
    return num == reversed
```

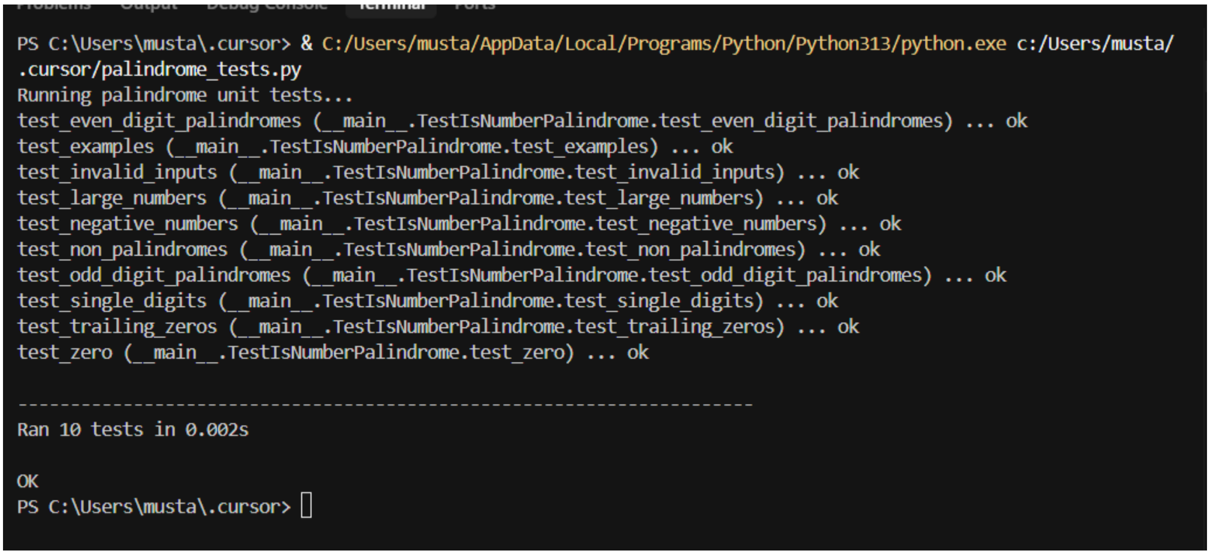


```
class TestIsNumberPalindrome(unittest.TestCase):
    def test_examples(self):
        self.assertTrue(is_number_palindrome(121))
        self.assertFalse(is_number_palindrome(123))

    def test_zero(self):
        self.assertTrue(is_number_palindrome(0))

    def test_single_digits(self):
        for d in range(10):
            self.assertTrue(is_number_palindrome(d))

    def test_negative_numbers(self):
        self.assertFalse(is_number_palindrome(-1))
        self.assertFalse(is_number_palindrome(-121))
        self.assertFalse(is_number_palindrome(-101))

    def test_trailing_zeros(self):
        self.assertFalse(is_number_palindrome(10))
        self.assertFalse(is_number_palindrome(100))
        self.assertFalse(is_number_palindrome(1000))

    def test_even_digit_palindromes(self):
        self.assertTrue(is_number_palindrome(1221))
        self.assertTrue(is_number_palindrome(3443))
        self.assertTrue(is_number_palindrome(9009))
        self.assertTrue(is_number_palindrome(1001))

    def test_odd_digit_pal
        self.assertTrue(is_
        self.assertFalse(is_number_palindrome(133))  # 133 is not a palindrome
```



```
class TestIsNumberPalindrome(unittest.TestCase):
    def test_odd_digit_palindromes(self):
        self.assertTrue(is_number_palindrome(12321))
        self.assertTrue(is_number_palindrome(1234321))

    def test_non_palindromes(self):
        self.assertFalse(is_number_palindrome(12))
        self.assertFalse(is_number_palindrome(1231))
        self.assertFalse(is_number_palindrome(123421))
        self.assertFalse(is_number_palindrome(1002))

    def test_large_numbers(self):
        self.assertTrue(is_number_palindrome(123454321))
        self.assertTrue(is_number_palindrome(1000000000000000001))
        self.assertFalse(is_number_palindrome(1234567890123456780))

    def test_invalid_inputs(self):
        self.assertFalse(is_number_palindrome(12.21))
        self.assertFalse(is_number_palindrome("121"))
        self.assertFalse(is_number_palindrome(None))
        self.assertFalse(is_number_palindrome(True))
        self.assertFalse(is_number_palindrome([1,2,1]))


if __name__ == "__main__":
    print("Running palindrome unit tests...")
    unittest.main(verbosity=2)
```

Output:



```
PS C:\Users\musta\.cursor> & C:/Users/musta/AppData/Local/Programs/Python/Python313/python.exe c:/Users/musta/
.cursor/palindrome_tests.py
Running palindrome unit tests...
test_even_digit_palindromes (__main__.TestIsNumberPalindrome.test_even_digit_palindromes) ... ok
test_examples (__main__.TestIsNumberPalindrome.test_examples) ... ok
test_invalid_inputs (__main__.TestIsNumberPalindrome.test_invalid_inputs) ... ok
test_large_numbers (__main__.TestIsNumberPalindrome.test_large_numbers) ... ok
test_negative_numbers (__main__.TestIsNumberPalindrome.test_negative_numbers) ... ok
test_non_palindromes (__main__.TestIsNumberPalindrome.test_non_palindromes) ... ok
test_odd_digit_palindromes (__main__.TestIsNumberPalindrome.test_odd_digit_palindromes) ... ok
test_single_digits (__main__.TestIsNumberPalindrome.test_single_digits) ... ok
test_trailing_zeros (__main__.TestIsNumberPalindrome.test_trailing_zeros) ... ok
test_zero (__main__.TestIsNumberPalindrome.test_zero) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.002s

OK
PS C:\Users\musta\.cursor> 
```

Observation:

All 10 unit tests for the palindrome program passed successfully with OK. The implementation handles valid, invalid, and edge cases correctly.