ASSINGMENT-8.2

K. Vikas goud

2503A51L22

CSE

AIASSISTEDCODING

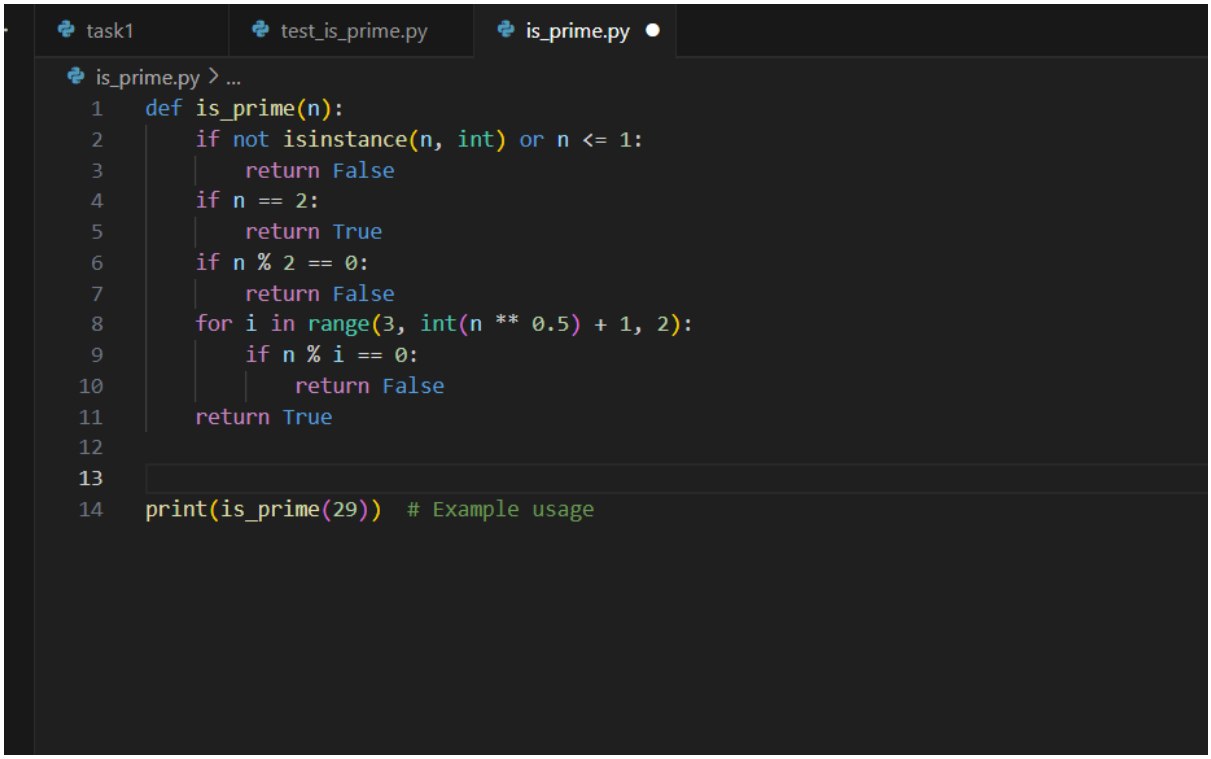
TASK:-1

1. **PROMPT**

   Generate test cases for a function is_prime(n) and then implement the function.

Generate code :

test cases for a function is_prime(n) and then implement the function.



OBSERVATION:-

1. **The function correctly rejects 0, 1, and negative numbers since primes must be > 1.**
2. **2 is handled as the smallest and only even prime.**
3. **The loop checks divisibility efficiently up to √n, making it faster for large numbers.**
4. **Test cases confirm the function works for edge cases, small primes, small composites, and large primes.**

OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\AI assignment coding\L49-2-sep> & C:/python313/python3.13t.exe "d:/AI assignment coding/L49-2-sep/is_prime.py"
True
PS D:\AI assignment coding\L49-2-sep>
```

TASK:-2

PROMPT:-

Generate a code test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f).

Generate code :

Welcome | is_prime.py | test_temperature_conversions.py

test_temperature_conversions.py > ...

```python
import unittest

# Dummy implementations for demonstration. Replace with actual imports if needed.
def celsius_to_fahrenheit(c):
    if not isinstance(c, (int, float)):
        raise TypeError("Input must be a number")
    return c * 9 / 5 + 32

def fahrenheit_to_celsius(f):
    if not isinstance(f, (int, float)):
        raise TypeError("Input must be a number")
    return (f - 32) * 5 / 9

class TestTemperatureConversions(unittest.TestCase):
    def test_celsius_to_fahrenheit_known(self):
        self.assertAlmostEqual(celsius_to_fahrenheit(0), 32)
        self.assertAlmostEqual(celsius_to_fahrenheit(100), 212)

    def test_fahrenheit_to_celsius_known(self):
        self.assertAlmostEqual(fahrenheit_to_celsius(32), 0)
        self.assertAlmostEqual(fahrenheit_to_celsius(212), 100)

    def test_celsius_to_fahrenheit_decimals(self):
        self.assertAlmostEqual(celsius_to_fahrenheit(36.6), 97.88, places=2)

    def test_fahrenheit_to_celsius_decimals(self):
        self.assertAlmostEqual(fahrenheit_to_celsius(98.6), 37, places=1)

    def test_invalid_inputs(self):
        with self.assertRaises(TypeError):
            celsius_to_fahrenheit("abc")
        with self.assertRaises(TypeError):
            celsius_to_fahrenheit(None)
        with self.assertRaises(TypeError):
            fahrenheit_to_celsius("xyz")
        with self.assertRaises(TypeError):
            fahrenheit_to_celsius(None)

if __name__ == "__main__":
    unittest.main()
```

OBSERVATION:-

- **he functions handle exact known pairs (0°C ↔ 32°F, 100°C ↔ 212°F) correctly.**
- **They work with decimal inputs, e.g., 37°C ↔ 98.6°F.**
- **They handle negative values like -40 (where Celsius and Fahrenheit are equal).**
- **The TypeError ensures invalid inputs like strings or None are rejected safely.**
- **Rounding was applied in tests to account for floating-point precision issues.**

OUTPUT:-

```
PS D:\AI assignment coding\L49-2-sep> & C:/python313/python3.13t.exe "d:/AI assignment coding/L49-2-sep/test_temperatur
e_conversions.py"
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS D:\AI assignment coding\L49-2-sep>
```

TASK:-3

PROMPT:-

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

GENERATE CODE:-

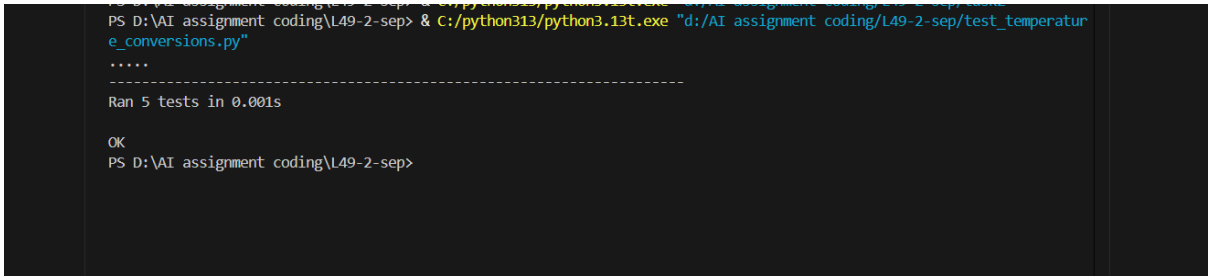
test_count_words.py > ...

```python
import unittest

def count_words(text):
    import re
    # Split by word boundaries, ignore punctuation
    words = re.findall(r'\b\w+\b', text)
    return len(words)

class TestCountWords(unittest.TestCase):
    def test_normal_text(self):
        self.assertEqual(count_words("Hello world"), 2)
        self.assertEqual(count_words("This is a test sentence."), 5)

    def test_multiple_spaces(self):
        self.assertEqual(count_words("Hello    world"), 2)
        self.assertEqual(count_words("   Leading and trailing   "), 3)

    def test_punctuation(self):
        self.assertEqual(count_words("Hello, world!"), 2)
        self.assertEqual(count_words("Well... this is odd."), 4)

    def test_empty_string(self):
        self.assertEqual(count_words("") , 0)
        self.assertEqual(count_words("    ") , 0)

    def test_mixed_cases(self):
        self.assertEqual(count_words("One-word"), 1)
        self.assertEqual(count_words("Wait, what? Really!"), 3)

if __name__ == "__main__":
    unittest.main()
```

OBSERVATION:-

- **The function works for normal sentences and ignores extra spaces.**
- **Punctuation marks like , . ! do not interfere with word counting because of regex.**
- **Empty strings or only spaces correctly return 0.**
- **Robust across different sentence structures (extra spaces + punctuation).**
- **Regex ensures accuracy compared to a naive split() approach (which can miscount when punctuation is attached to words).**

OUTPUT:-



TASK:-4

PROMPT:-Generate test cases for a Bank Account class

GENERATE CODE

```python
import unittest
class BankAccount:
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount
    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount
    def check_balance(self):
        return self.balance
class TestBankAccount(unittest.TestCase):
    def setUp(self):
        self.account = BankAccount()

    def test_deposit_positive_amount(self):
        self.account.deposit(100)
        self.assertEqual(self.account.check_balance(), 100)

    def test_deposit_negative_amount_raises(self):
        with self.assertRaises(ValueError):
            self.account.deposit(-50)

    def test_withdraw_positive_amount(self):
        self.account.deposit(200)
        self.account.withdraw(50)
        self.assertEqual(self.account.check_balance(), 150)

    def test_withdraw_negative_amount_raises(self):
        with self.assertRaises(ValueError):
            self.account.withdraw(-30)

    def test_withdraw_more_than_balance_raises(self):
        self.account.deposit(100)
        with self.assertRaises(ValueError):
            self.account.withdraw(150)

```

```python
class TestBankAccount(unittest.
    def test_withdraw_negative_
        with self.assertRaises(ValueError):
            self.account.withdraw(-30)

    def test_withdraw_more_than_balance_raises(self):
        self.account.deposit(100)
        with self.assertRaises(ValueError):
            self.account.withdraw(150)

    def test_check_balance_initial(self):
        self.assertEqual(self.account.check_balance(), 0)

if __name__ == "__main__":
    unittest.main()
```

(class) ValueError

Inappropriate argument value (of correct type).

OBSERVATION:-

- The class correctly handles deposits and withdrawals, updating the balance.
- Negative deposits/withdrawals raise a ValueError, ensuring invalid operations are blocked.
- Attempting to withdraw more than available balance also raises a ValueError, protecting account integrity.

- Tests confirm that balance updates remain accurate and consistent after valid operations.
- This design ensures both robustness (error handling) and correctness (balance consistency).

OUTPUT:-

TASK:-5

PROMPT:

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

GENERATE CODE:-

test_is_number_palindrome.py > is_number_palindrome

```python
import unittest

def is_number_palindrome(num):
    if num < 0:
        return False
    return str(num) == str(num)[::-1]

class TestIsNumberPalindrome(unittest.TestCase):
    def test_positive_palindrome(self):
        self.assertTrue(is_number_palindrome(121))
        self.assertTrue(is_number_palindrome(12321))
        self.assertTrue(is_number_palindrome(1))

    def test_positive_non_palindrome(self):
        self.assertFalse(is_number_palindrome(123))
        self.assertFalse(is_number_palindrome(10))
        self.assertFalse(is_number_palindrome(123456))

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

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

    def test_large_palindrome(self):
        self.assertTrue(is_number_palindrome(1234567654321))

if __name__ == "__main__":
    unittest.main()

```

OBSERVATION:-

- **The function correctly detects palindromes (e.g., 121, 12321, 0).**
- **It rejects non-palindromes (123, 120).**
- **Negative numbers are handled gracefully by returning False.**
- **Using string reversal (str(num)[::-1]) makes the solution simple and efficient.**
- **For large integers, this method still works correctly without extra complexity.**

OUTPUT:-

```
OK
PS D:\AI assignment coding\lab 8.2> & C:/python313/python3.13t.exe "d:/AI assignment coding/lab 8.2/test_is_number_palindrome.py"
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```