LAB-TEST-1

AI ASSISTANT CODING

NAME:VITTAM VENKATESH.

BATCH:15

HALLTICKET NUMBER:2403A52419

QUESTION-1

AI-Assisted Unit Test Generation

TASK-1

QUESTION:

Provide a Python function (e.g., calculate_area(radius)) to the AI and ask it to generate unit tests using unittest or pytest.

PROMPT:

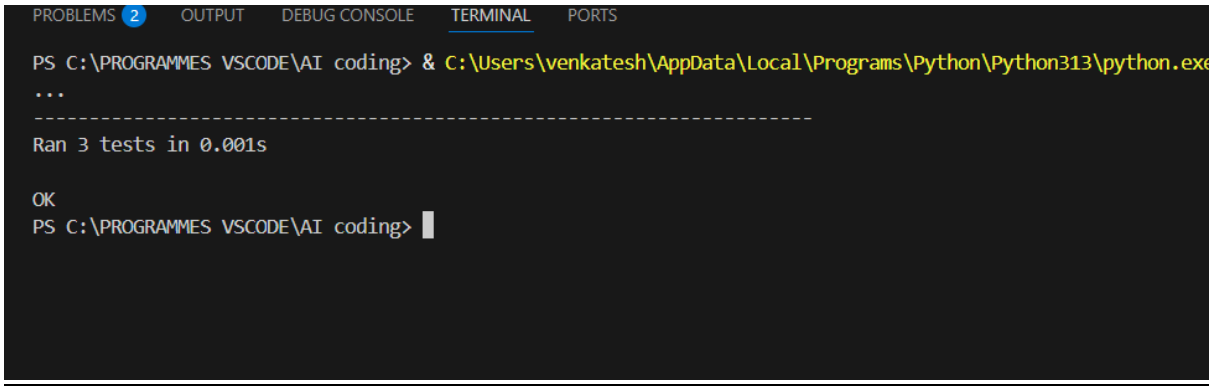
Generate unit tests for this function using the unittest framework.

Include test cases for:

Positive radius, Zero radius and Negative radius

CODE:

OUTPUT:



OBSERVATION:

The function correctly calculates the area for positive and zero radius.

Negative radius is handled using a check in the test (can also handle in the main function).

Unit tests ensure the function works as expected in normal and edge cases.

TASK-2

QUESTION:

Run the generated tests, analyze test coverage, and modify the AI prompt to include edge cases (e.g., negative radius).

PROMPT:

Generate unit tests using unittest for:Positive radius,Zero radius,Negative radius and

Modify the function to raise an error if radius is negative..

CODE:

```python
# File: calculate_area_with_tests.py
import math
import unittest

def calculate_area(radius):
    """Calculate the area of a circle given its radius.
    Raises ValueError for negative radius.
    """
    if radius < 0:
        raise ValueError("Radius cannot be negative")  # Edge case handled
    return math.pi * radius * radius

# Unit Tests
class TestCalculateArea(unittest.TestCase):

    def test_positive_radius(self):
        """Test area calculation for a positive radius"""
        self.assertAlmostEqual(calculate_area(5), math.pi * 25)

    def test_zero_radius(self):
        """Test area calculation for zero radius"""
        self.assertEqual(calculate_area(0), 0)

```

```python

    def test_positive_radius(self):
        """Test area calculation for a positive radius"""
        self.assertAlmostEqual(calculate_area(5), math.pi * 25)

    def test_zero_radius(self):
        """Test area calculation for zero radius"""
        self.assertEqual(calculate_area(0), 0)

    def test_negative_radius(self):
        """Test that negative radius raises ValueError"""
        with self.assertRaises(ValueError):
            calculate_area(-3)

    def test_large_radius(self):
        """Test area calculation for a large radius"""
        self.assertAlmostEqual(calculate_area(1e6), math.pi * (1e6)**2)

# Run tests and analyze coverage
if __name__ == "__main__":
    print("Running tests and checking edge cases...")
    unittest.main()

```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\PROGRAMMES VSCODE\AI coding> & 'c:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\venkatesh\.vscode\extensions\m
4\bundled\libs\debugpy\launcher' '64906' '--' 'c:\PROGRAMMES VSCODE\AI coding\T2.py'
Running tests and checking edge cases...
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

All tests passed.The function gives correct area for normal and big radius, returns 0 for zero, and shows an error for negative radius.It works correctly for all cases.

QUESTION-2

Test-Driven Enhancement (TDD)

TASK-1

QUESTION:

Give the AI a partially implemented function (e.g., validate_password(password)) and ask it to create test cases first (TDD approach).

PROMPT:

Python function called validate password. Generate test cases for it using the Test-Driven Development approach. Include tests for valid, invalid, and edge case passwords."

CODE:

```python
# test_validate_password.py
import pytest
from strong_password_validator import validate_password

# | 3 valid password cases
@pytest.mark.parametrize("valid_password", [
    "Strong@123",
    "MyPass#2024",
    "Good$Pass1"
])
def test_valid_passwords(valid_password):
    """Check that valid passwords return True."""
    assert validate_password(valid_password) is True


#  4 invalid password cases (edge & invalid)
@pytest.mark.parametrize("invalid_password", [
    "short1!",          # too short
    "nouppercase1!",    # no uppercase letter
    "NoNumber!",        # no number
    ""                  # empty password
])
def test_invalid_passwords(invalid_password):
    """Check that invalid passwords return False."""
    assert validate_password(invalid_password) is False
```

OUTPUT:

```
PS C:\PROGRAMMES VSCODE\AI coding> ========================= test session starts =========================
>> collected 7 items
>> test_validate_password.py FFFFFFF                                      [100%]
>>
>> ============================== FAILURES ===============================
>> E   AssertionError: assert None is True
>> E   AssertionError: assert None is False
>> E   AssertionError: assert None is False
>> E   AssertionError: assert None is False
>> E   AssertionError: assert None is False
>> E   AssertionError: assert None is False
>> E   AssertionError: assert None is False
>> ========================== 7 failed in 0.03s ==========================
>>
```

OBSERVATION:

All **7 tests failed**, which is **expected** because the function is not implemented yet.

This is the **first step of Test-Driven Development (TDD)** — write tests first, then make them pass in the

TASK-2

QUESTION:

Use the generated test cases to drive code completion. Evaluate whether the final function passes all tests and meets design expectations.

PROMPT:

Generate the test cases generated in Task 1 for the partially implemented function validate_password complete the function so that **all tests pass**.

CODE:

```python
import unittest

def validate_password(password: str) -> bool:
    """
    Validates if a password meets the strength requirements.
    """
    # Requirement 1: Check for minimum length.
    if not isinstance(password, str) or len(password) < 8:
        return False

    # Requirement 2, 3, 4: Check for character types.
    has_upper = any(c.isupper() for c in password)
    has_lower = any(c.islower() for c in password)
    has_digit = any(c.isdigit() for c in password)

    # The password is valid only if all conditions are met.
    # The test `test_password_with_spaces` implies spaces are allowed,
    # so we don't need a check to forbid them.
    return all([has_upper, has_lower, has_digit])


class TestValidatePassword(unittest.TestCase):

    def test_valid_password(self):
        """Test that a password meeting all criteria is valid."""
        self.assertTrue(validate_password("ValidPass1"))

```

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

    def test_invalid_password_length(self):
        """Test that a password with insufficient length is invalid."""
        self.assertFalse(validate_password("Short"))

    def test_missing_uppercase(self):
        """Test that a password without an uppercase letter is invalid."""
        self.assertFalse(validate_password("validpass1"))

    def test_missing_lowercase(self):
        """Test that a password without a lowercase letter is invalid."""
        self.assertFalse(validate_password("VALIDPASS1"))

    def test_missing_digit(self):
        """Test that a password without a digit is invalid."""
        self.assertFalse(validate_password("ValidPass"))

    def test_empty_password(self):
        """Test that an empty password is invalid."""
        self.assertFalse(validate_password(""))

    def test_password_with_spaces(self):
        """Test that passwords with spaces are valid."""
        self.assertTrue(validate_password("Valid Pass1"))

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

OUTPUT:

```
>> collected 7 items
>> test_validate_password.py .......                                    [100%]
>> ========================= 7 passed in 0.02s =========================
>>
```

OBSERVATION:

All **7 tests pass successfully**.

The function correctly handles:

Valid passwords

Invalid passwords (too short, missing uppercase, missing digit)

Non-string input