

Idm:to20raje

[https://github.com/Rutvi1309/dsss\\_homework\\_2](https://github.com/Rutvi1309/dsss_homework_2)

### Task:3

```
import random

def generate_random_integer(min_value, max_value):
    """Generate a random integer within the given range."""
    return random.randint(min_value, max_value)

def generate_operator():
    """Generate a random arithmetic operator: +, -, *, /."""
    return random.choice(['+', '-', '*', '/'])

def calculate_result(num1, num2, operator):
    """
    Calculate the result of an arithmetic operation and return both the problem and the answer.
    """
    problem = f"{num1} {operator} {num2}"
    if operator == '+':
        answer = num1 + num2
    elif operator == '-':
        answer = num1 - num2
    else:
        answer = num1 * num2
    return problem, answer
```

```
def math_quiz():
    score = 0
    total_questions = 3 # You can change the number of questions here

    print("Welcome to the Math Quiz Game!")
    print("You will be presented with math problems, and you need to provide the correct answers.")

    for i in range(total_questions):
        num1 = generate_random_integer(1, 10)
        num2 = generate_random_integer(1, 5) # Adjusted max value to an integer
        operator = generate_operator()

        problem, answer = calculate_result(num1, num2, operator)
        print(f"Question: {problem}")

        # Error handling for non-integer inputs
        while True:
            try:
                user_answer = int(input("Your answer: "))
                break # Break the loop if input is an integer
            except ValueError:
                print("Please enter a valid integer.")

        if user_answer == answer:
            print("Correct! You earned a point.")
            score += 1
        else:
            print(f"Wrong answer. The correct answer is {answer}.")

    print(f"Game over! Your score is: {score}/{total_questions}")
```

```
PS C:\Users\rutvi\dss homework_2\math_quiz> git push origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.06 KiB | 1.06 MiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/Rutvi1309/dss homework_2.git
9fad084..957b0b7 main -> main
```

### Task:4

```
import unittest
from math import import generate_random_integer, generate_operator, calculate_result

class TestMathFunctions(unittest.TestCase):
    def test_generate_integer_within_range(self):
        min_val = 1
        max_val = 10
        rand_num = generate_random_integer(min_val, max_val)
        self.assertTrue(min_val <= rand_num <= max_val)

    def test_generate_operator(self):
        # Test if operator generated is one of the valid arithmetic operators
        valid_operators = '+', '-', '*', '/'
        operator = generate_operator()
        self.assertIn(operator, valid_operators)

    def test_calculate_result(self):
        # Test cases for the calculate_result function
        # Define some test cases to check if the calculations are correct
        test_cases = [
            (1, 2, '+', 3), # Simple addition
            (5, 2, '-', 3), # Simple subtraction
            (2, 3, '*', 6), # Simple multiplication
            (6, 2, '/', 3), # Simple division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_floats(self):
        # Test cases for the calculate_result function with floats
        test_cases = [
            (1.5, 2.5, '+', 4.0), # Addition with floats
            (5.5, 2.5, '-', 3.0), # Subtraction with floats
            (2.0, 3.0, '*', 6.0), # Multiplication with floats
            (6.0, 2.0, '/', 3.0), # Division with floats
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_negative_numbers(self):
        # Test cases for the calculate_result function with negative numbers
        test_cases = [
            (-1, 2, '+', 1), # Addition with negative
            (1, -2, '-', 3), # Subtraction with negative
            (-2, 3, '*', -6), # Multiplication with negative
            (6, -2, '/', -3), # Division with negative
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_parentheses(self):
        # Test cases for the calculate_result function with parentheses
        test_cases = [
            (1, 2, '(', 1), # Opening parenthesis
            (1, 2, ')', 2), # Closing parenthesis
            (1, 2, '(', 1), # Opening parenthesis
            (1, 2, ')', 2), # Closing parenthesis
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_invalid_input(self):
        # Test cases for the calculate_result function with invalid input
        test_cases = [
            (1, 2, '%', ValueError), # Modulo operator
            (1, 2, '^', ValueError), # Power operator
            (1, 2, '!', ValueError), # Factorial operator
            (1, 2, '@', ValueError), # At symbol
            (1, 2, '#', ValueError), # Hash symbol
            (1, 2, '$', ValueError), # Dollar sign
            (1, 2, '%', ValueError), # Modulo operator
            (1, 2, '^', ValueError), # Power operator
            (1, 2, '!', ValueError), # Factorial operator
            (1, 2, '@', ValueError), # At symbol
            (1, 2, '#', ValueError), # Hash symbol
            (1, 2, '$', ValueError), # Dollar sign
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_large_numbers(self):
        # Test cases for the calculate_result function with large numbers
        test_cases = [
            (1000000000, 1000000000, '+', 2000000000), # Large addition
            (1000000000, 1000000000, '-', 0), # Large subtraction
            (1000000000, 1000000000, '*', 1000000000000000), # Large multiplication
            (1000000000, 1000000000, '/', 1), # Large division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_small_numbers(self):
        # Test cases for the calculate_result function with small numbers
        test_cases = [
            (0.000000001, 0.000000001, '+', 2e-09), # Small addition
            (0.000000001, 0.000000001, '-', 0), # Small subtraction
            (0.000000001, 0.000000001, '*', 1e-18), # Small multiplication
            (0.000000001, 0.000000001, '/', 1), # Small division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_numbers(self):
        # Test cases for the calculate_result function with complex numbers
        test_cases = [
            (1j, 1j, '+', 2j), # Complex addition
            (1j, 1j, '-', 0j), # Complex subtraction
            (1j, 1j, '*', -1), # Complex multiplication
            (1j, 1j, '/', -1j), # Complex division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_infinity(self):
        # Test cases for the calculate_result function with infinity
        test_cases = [
            (float('inf'), 1, '+', float('inf')), # Infinity addition
            (float('inf'), 1, '-', float('inf')), # Infinity subtraction
            (float('inf'), 1, '*', float('inf')), # Infinity multiplication
            (float('inf'), 1, '/', float('inf')), # Infinity division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_nan(self):
        # Test cases for the calculate_result function with NaN
        test_cases = [
            (float('nan'), 1, '+', float('nan')), # NaN addition
            (float('nan'), 1, '-', float('nan')), # NaN subtraction
            (float('nan'), 1, '*', float('nan')), # NaN multiplication
            (float('nan'), 1, '/', float('nan')), # NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_infinity(self):
        # Test cases for the calculate_result function with complex and infinity
        test_cases = [
            (1j, float('inf'), '+', float('inf')), # Complex and infinity addition
            (1j, float('inf'), '-', float('inf')), # Complex and infinity subtraction
            (1j, float('inf'), '*', float('inf')), # Complex and infinity multiplication
            (1j, float('inf'), '/', float('inf')), # Complex and infinity division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_nan(self):
        # Test cases for the calculate_result function with complex and NaN
        test_cases = [
            (1j, float('nan'), '+', float('nan')), # Complex and NaN addition
            (1j, float('nan'), '-', float('nan')), # Complex and NaN subtraction
            (1j, float('nan'), '*', float('nan')), # Complex and NaN multiplication
            (1j, float('nan'), '/', float('nan')), # Complex and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_infinity_and_nan(self):
        # Test cases for the calculate_result function with infinity and NaN
        test_cases = [
            (float('inf'), float('nan'), '+', float('nan')), # Infinity and NaN addition
            (float('inf'), float('nan'), '-', float('nan')), # Infinity and NaN subtraction
            (float('inf'), float('nan'), '*', float('nan')), # Infinity and NaN multiplication
            (float('inf'), float('nan'), '/', float('nan')), # Infinity and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_complex(self):
        # Test cases for the calculate_result function with complex and complex
        test_cases = [
            (1j, 1j, '+', 2j), # Complex and complex addition
            (1j, 1j, '-', 0j), # Complex and complex subtraction
            (1j, 1j, '*', -1), # Complex and complex multiplication
            (1j, 1j, '/', -1j), # Complex and complex division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_infinity_and_infinity(self):
        # Test cases for the calculate_result function with infinity and infinity
        test_cases = [
            (float('inf'), float('inf'), '+', float('inf')), # Infinity and infinity addition
            (float('inf'), float('inf'), '-', float('inf')), # Infinity and infinity subtraction
            (float('inf'), float('inf'), '*', float('inf')), # Infinity and infinity multiplication
            (float('inf'), float('inf'), '/', float('inf')), # Infinity and infinity division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_nan_and_nan(self):
        # Test cases for the calculate_result function with NaN and NaN
        test_cases = [
            (float('nan'), float('nan'), '+', float('nan')), # NaN and NaN addition
            (float('nan'), float('nan'), '-', float('nan')), # NaN and NaN subtraction
            (float('nan'), float('nan'), '*', float('nan')), # NaN and NaN multiplication
            (float('nan'), float('nan'), '/', float('nan')), # NaN and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_infinity_and_nan(self):
        # Test cases for the calculate_result function with complex, infinity, and NaN
        test_cases = [
            (1j, float('inf'), float('nan'), '+', float('nan')), # Complex, infinity, and NaN addition
            (1j, float('inf'), float('nan'), '-', float('nan')), # Complex, infinity, and NaN subtraction
            (1j, float('inf'), float('nan'), '*', float('nan')), # Complex, infinity, and NaN multiplication
            (1j, float('inf'), float('nan'), '/', float('nan')), # Complex, infinity, and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_infinity_and_infinity_and_nan(self):
        # Test cases for the calculate_result function with infinity, infinity, and NaN
        test_cases = [
            (float('inf'), float('inf'), float('nan'), '+', float('nan')), # Infinity, infinity, and NaN addition
            (float('inf'), float('inf'), float('nan'), '-', float('nan')), # Infinity, infinity, and NaN subtraction
            (float('inf'), float('inf'), float('nan'), '*', float('nan')), # Infinity, infinity, and NaN multiplication
            (float('inf'), float('inf'), float('nan'), '/', float('nan')), # Infinity, infinity, and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_complex_and_infinity_and_nan(self):
        # Test cases for the calculate_result function with complex, complex, infinity, and NaN
        test_cases = [
            (1j, 1j, float('inf'), float('nan'), '+', float('nan')), # Complex, complex, infinity, and NaN addition
            (1j, 1j, float('inf'), float('nan'), '-', float('nan')), # Complex, complex, infinity, and NaN subtraction
            (1j, 1j, float('inf'), float('nan'), '*', float('nan')), # Complex, complex, infinity, and NaN multiplication
            (1j, 1j, float('inf'), float('nan'), '/', float('nan')), # Complex, complex, infinity, and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_infinity_and_infinity_and_infinity_and_nan(self):
        # Test cases for the calculate_result function with infinity, infinity, infinity, and NaN
        test_cases = [
            (float('inf'), float('inf'), float('inf'), float('nan'), '+', float('nan')), # Infinity, infinity, infinity, and NaN addition
            (float('inf'), float('inf'), float('inf'), float('nan'), '-', float('nan')), # Infinity, infinity, infinity, and NaN subtraction
            (float('inf'), float('inf'), float('inf'), float('nan'), '*', float('nan')), # Infinity, infinity, infinity, and NaN multiplication
            (float('inf'), float('inf'), float('inf'), float('nan'), '/', float('nan')), # Infinity, infinity, infinity, and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_complex_and_infinity_and_infinity_and_nan(self):
        # Test cases for the calculate_result function with complex, complex, infinity, infinity, and NaN
        test_cases = [
            (1j, 1j, float('inf'), float('inf'), float('nan'), '+', float('nan')), # Complex, complex, infinity, infinity, and NaN addition
            (1j, 1j, float('inf'), float('inf'), float('nan'), '-', float('nan')), # Complex, complex, infinity, infinity, and NaN subtraction
            (1j, 1j, float('inf'), float('inf'), float('nan'), '*', float('nan')), # Complex, complex, infinity, infinity, and NaN multiplication
            (1j, 1j, float('inf'), float('inf'), float('nan'), '/', float('nan')), # Complex, complex, infinity, infinity, and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_infinity_and_infinity_and_infinity_and_infinity_and_nan(self):
        # Test cases for the calculate_result function with infinity, infinity, infinity, infinity, and NaN
        test_cases = [
            (float('inf'), float('inf'), float('inf'), float('inf'), float('nan'), '+', float('nan')), # Infinity, infinity, infinity, infinity, and NaN addition
            (float('inf'), float('inf'), float('inf'), float('inf'), float('nan'), '-', float('nan')), # Infinity, infinity, infinity, infinity, and NaN subtraction
            (float('inf'), float('inf'), float('inf'), float('inf'), float('nan'), '*', float('nan')), # Infinity, infinity, infinity, infinity, and NaN multiplication
            (float('inf'), float('inf'), float('inf'), float('inf'), float('nan'), '/', float('nan')), # Infinity, infinity, infinity, infinity, and NaN division
        ]
        for (a, b, op, expected) in test_cases:
            result = calculate_result(a, b, op)
            self.assertEqual(result, expected)

    def test_calculate_result_with_complex_and_complex_and_infinity_and_infinity_and_infinity_and_nan(self):

```

### Task:5

[illegible]

