МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра інформаційних систем та мереж

Лабораторна робота №6

з дисципліни

СПЕЦІАЛІЗОВАНІ МОВИ ПРОГРАМУВАННЯ

на тему

Розробка та Unit тестування Python додатку

Виконав:

ст. гр. РІ-21сп

Павлусь В.П

Прийняв:

Шербак С.С.

Львів-2024

**Мета лабораторної роботи:** Cтворення юніт-тестів для додатка-калькулятора на основі класів

**Хід роботи**

**Завдання 1: Тестування Додавання.**

Напишіть юніт-тест, щоб перевірити, що операція додавання в вашому додатку-калькуляторі працює правильно. Надайте тестові випадки як для позитивних, так і для негативних чисел.

**Завдання 2: Тестування Віднімання.**

Створіть юніт-тести для переконання, що операція віднімання працює правильно. Тестуйте різні сценарії, включаючи випадки з від'ємними результатами.

**Завдання 3: Тестування Множення.**

Напишіть юніт-тести, щоб перевірити правильність операції множення в вашому калькуляторі. Включіть випадки з нулем, позитивними та від'ємними числами.

**Завдання 4: Тестування Ділення.**

Розробіть юніт-тести для підтвердження точності операції ділення. Тести повинні охоплювати ситуації, пов'язані з діленням на нуль та різними числовими значеннями.

**Завдання 5: Тестування Обробки Помилок.**

Створіть юніт-тести, щоб перевірити, як ваш додаток-калькулятор обробляє помилки. Включіть тести для ділення на нуль та інших потенційних сценаріїв помилок. Переконайтеся, що додаток відображає відповідні повідомлення про помилки.

**Виконання роботи**

**Вміст файлу classes.py:**

import math

from abc import ABC, abstractmethod

from variables import memory, history, decimal\_places

class Memory:

def \_\_init\_\_(self):

self.\_\_memory = memory

def set\_memory(self, value):

self.\_\_memory = value

def get\_memory(self):

return self.\_\_memory

def clear\_memory(self):

self.\_\_memory = 0

class History:

def \_\_init\_\_(self):

self.\_\_history = history

def add\_to\_history(self, expression, result):

self.\_\_history.append(f"{expression} = {result}")

def show\_history(self):

if self.\_\_history:

print("Calculating history:")

for entry in self.\_\_history:

print(entry)

else:

print("History is empty.")

class Settings:

def \_\_init\_\_(self):

self.\_\_decimal\_places = decimal\_places

def set\_decimal\_places(self):

places = input("Enter the number of symbols after the decimal point: ")

if places.isalpha():

raise TypeError("Please, enter a positive integer!")

elif int(places) >= 0:

self.\_\_decimal\_places = int(places)

print(f"Number of symbols after decimal point - {self.\_\_decimal\_places}.")

else:

raise ValueError("Please, try again. Number of symbols must not be negated!")

def get\_decimal\_places(self):

return self.\_\_decimal\_places

class BaseCalculator(ABC):

def \_\_init\_\_(self):

self.memory = Memory()

self.history = History()

self.settings = Settings()

def get\_input(self):

try:

num1\_input = input("Enter first number or 'm' for use value from memory: ")

num1 = self.memory.get\_memory() if num1\_input.lower() == 'm' else float(num1\_input)

operator = self.get\_operator()

num2 = None

if operator != '√':

while True:

try:

num2\_input = input("Enter second number or 'm' for use value from memory: ")

num2 = self.memory.get\_memory() if num2\_input.lower() == 'm' else float(num2\_input)

break

except ValueError:

print("Incorrect input! Please, try again!")

return num1, operator, num2

except ValueError:

print("Incorrect input! Please, try again!")

return self.get\_input()

def check\_operator(self, operator):

return operator in ['+', '-', '\*', '/', '^', '√', '%']

def get\_operator(self):

while True:

operator = input("Choose one of the operators (+, -, \*, /, ^, √, %): ")

if self.check\_operator(operator):

return operator

else:

print("Incorrect operator! Please, try again!")

@abstractmethod

def perform\_operators(self, num1, operator, num2):

pass

class Calculator(BaseCalculator):

def perform\_operators(self, num1, operator, num2):

decimal\_places = self.settings.get\_decimal\_places()

match operator:

case '+':

return round(num1 + num2, decimal\_places)

case '-':

return round(num1 - num2, decimal\_places)

case '\*':

return round(num1 \* num2, decimal\_places)

case '/':

if num2 == 0:

raise ZeroDivisionError("You can't divide by zero!")

return round(num1 / num2, decimal\_places)

case '^':

result = math.pow(abs(num1), num2)

if num1 < 0:

result = -result

return round(result, decimal\_places)

case '√':

if num1 < 0:

raise ValueError("You can't get a square root from a negative number!")

return round(math.sqrt(num1), decimal\_places)

case '%':

if num2 == 0:

raise ZeroDivisionError("You can't get a percentage from zero!")

return round(num1 % num2, decimal\_places)

case \_:

raise ValueError("Incorrect operator!")

def calculate(self):

while True:

num1, operator, num2 = self.get\_input()

result = self.perform\_operators(num1, operator, num2)

if result is not None:

print(f"Result: {result}")

self.history.add\_to\_history(f"{num1} {operator} {num2 if num2 is not None else ''}", result)

if input("Do you want to save result? (y/n): ").lower() == 'y':

self.memory.set\_memory(result)

print(f"Result {result} was saved.")

if input("Do you want to do new calculation? (y/n): ").lower() == 'n':

break

**Вміст файлу interface.py:**

from classes import Calculator

def main():

try:

calc = Calculator()

except TypeError as e:

print(e)

exit(1)

while True:

print("\nCalculator")

print("1. Perform calculations.")

print("2. Show history.")

print("3. Get result from memory.")

print("4. Clean memory.")

print("5. Change the number of symbols after the decimal point.")

print("6. Exit.")

choice = input("Select option: ")

match choice:

case '1':

try:

calc.calculate()

except ZeroDivisionError as e:

print(e)

except ValueError as e:

print(e)

case '2':

calc.history.show\_history()

case '3':

print(f"Saved value: {calc.memory.get\_memory()}")

case '4':

calc.memory.clear\_memory()

print("Saved value cleared.")

case '5':

try:

calc.settings.set\_decimal\_places()

except ValueError as e:

print(e)

except TypeError as e:

print(e)

case '6':

print("Exit from calculator.")

break

case \_:

print("Wrong choice! Please try again!")

**Вміст файлу run\_tests\_with\_coverage.py:**

import coverage

import unittest

def main():

cov = coverage.Coverage()

cov.start()

loader = unittest.TestLoader()

tests = loader.discover('.')

test\_runner = unittest.TextTestRunner()

result = test\_runner.run(tests)

cov.stop()

cov.save()

print("\n--- Code Coverage Report ---")

cov.report()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Вміст файлу test\_calculator.py:**

import unittest

from unittest.mock import patch

from classes import Calculator

class TestCalculator(unittest.TestCase):

def setUp(self):

self.calc = Calculator()

# Тест перевіряє правильне введення чисел і оператора

@patch('builtins.input', side\_effect=['10', '+', '1'])

def test\_get\_input\_valid(self, mock\_input):

num1, operator, num2 = self.calc.get\_input()

self.assertEqual(num1, 10.0)

self.assertEqual(operator, '+')

self.assertEqual(num2, 1.0)

# Тест перевіряє використання пам'яті та оператора квадратного кореня

@patch('builtins.input', side\_effect=['m', '√'])

def test\_get\_input\_memory\_sqrt(self, mock\_input):

self.calc.memory.set\_memory(16.0)

num1, operator, num2 = self.calc.get\_input()

self.assertEqual(num1, 16.0)

self.assertEqual(operator, '√')

self.assertIsNone(num2)

# Тест перевіряє обробку неправильного оператора та повторне запитання оператора

@patch('builtins.input', side\_effect=['$', '+'])

def test\_get\_operator\_invalid\_then\_valid(self, mock\_input):

operator = self.calc.get\_operator()

self.assertEqual(operator, '+')

# Тест перевіряє правильність розпізнавання коректних операторів

def test\_check\_operator\_valid(self):

self.assertTrue(self.calc.check\_operator('+'))

self.assertTrue(self.calc.check\_operator('-'))

self.assertTrue(self.calc.check\_operator('\*'))

self.assertTrue(self.calc.check\_operator('/'))

self.assertTrue(self.calc.check\_operator('^'))

self.assertTrue(self.calc.check\_operator('√'))

self.assertTrue(self.calc.check\_operator('%'))

# Тест перевіряє розпізнавання некоректних операторів

def test\_check\_operator\_invalid(self):

self.assertFalse(self.calc.check\_operator('&'))

self.assertFalse(self.calc.check\_operator('!'))

self.assertFalse(self.calc.check\_operator('incorrect'))

self.assertFalse(self.calc.check\_operator('1'))

# Тест перевіряє встановлення та отримання значення з пам'яті

def test\_memory\_set\_get(self):

self.calc.memory.set\_memory(10)

self.assertEqual(self.calc.memory.get\_memory(), 10)

# Тест перевіряє очищення пам'яті

def test\_memory\_clear(self):

self.calc.memory.set\_memory(10)

self.calc.memory.clear\_memory()

self.assertEqual(self.calc.memory.get\_memory(), 0)

# Тест перевіряє додавання записів в історію та перевірку наявності запису

def test\_history\_add\_and\_show(self):

self.calc.history.add\_to\_history("2 + 2", 4)

self.assertIn("2 + 2 = 4", self.calc.history.\_History\_\_history)

# Тест перевіряє встановлення кількості знаків після коми

def test\_set\_decimal\_places(self):

with patch('builtins.input', side\_effect=['2']):

self.calc.settings.set\_decimal\_places()

self.assertEqual(self.calc.settings.get\_decimal\_places(), 2)

# Тест перевіряє обробку неправильного введення для кількості знаків після коми

def test\_set\_decimal\_places\_invalid(self):

with patch('builtins.input', side\_effect=['-1']):

with self.assertRaises(ValueError):

self.calc.settings.set\_decimal\_places()

# Тести для базових операцій: додавання, віднімання, множення, ділення

def test\_addition(self):

self.assertEqual(self.calc.perform\_operators(2, '+', 2), 4)

self.assertEqual(self.calc.perform\_operators(-2, '+', -2), -4)

self.assertEqual(self.calc.perform\_operators(-2, '+', 2), 0)

def test\_subtraction(self):

self.assertEqual(self.calc.perform\_operators(2, '-', 2), 0)

self.assertEqual(self.calc.perform\_operators(-2, '-', -2), 0)

self.assertEqual(self.calc.perform\_operators(2, '-', 4), -2)

def test\_multiplication(self):

self.assertEqual(self.calc.perform\_operators(2, '\*', 2), 4)

self.assertEqual(self.calc.perform\_operators(2, '\*', 0), 0)

self.assertEqual(self.calc.perform\_operators(-2, '\*', 2), -4)

def test\_division(self):

self.assertEqual(self.calc.perform\_operators(4, '/', 2), 2)

with self.assertRaises(ZeroDivisionError):

self.calc.perform\_operators(2, '/', 0)

# Тест для операції квадратного кореня

def test\_square\_root(self):

self.assertEqual(self.calc.perform\_operators(4, '√', None), 2)

with self.assertRaises(ValueError):

self.calc.perform\_operators(-4, '√', None)

# Тест для операції остачі від ділення (modulo)

def test\_modulo(self):

self.assertEqual(self.calc.perform\_operators(5, '%', 3), 2)

with self.assertRaises(ZeroDivisionError):

self.calc.perform\_operators(1, '%', 0)

# Тести для операції піднесення до степеня (exponentiation)

def test\_exponentiation(self):

self.assertEqual(self.calc.perform\_operators(2, '^', 2), 4)

self.assertEqual(self.calc.perform\_operators(2, '^', 0), 1)

self.assertEqual(self.calc.perform\_operators(-2, '^', 2), -4)

self.assertEqual(self.calc.perform\_operators(2, '^', -1), 0.5)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Вміст файлу variables.py:**

memory = 0

history = []

decimal\_places = 2

**Вміст файл runner.py:**

from interface import main

if \_\_name\_\_ == "\_\_main\_\_":

main()

Результатом виконання є програма, яка здійснює Unit тестування Python додатку. Результат зображено на рис. 1.

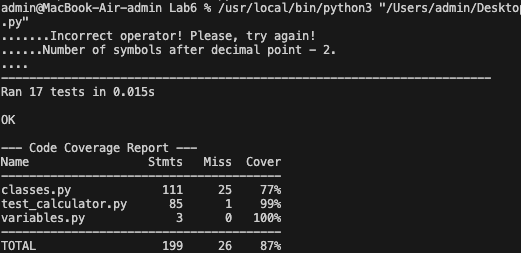


Рис. 1 – Результат Unit тестування

**Висновок:** У ході виконання ЛР я створив набір юніт-тестів, які перевіряють правильність основних арифметичних операцій у додатку-калькуляторі. Ці тести допомагають виявити та виправити будь-які проблеми або помилки, які можуть виникнути під час розробки чи обслуговування додатку, забезпечуючи його надійність і точність.