Звіт

З лабораторної роботи №2

Студента групи МІТ-31

Добровольського Арсенія Михайловича

Варіант №9

Тема роботи: Тестування програмного забезпечення

Мета роботи: розглянути принципи модульного тестування програмного забезпечення; навчитися створювати модульні тести з використанням unittest.

Посилання на Github репозиторій: https://github.com/arsgooo/pt2023

Пояснення до коду

Створюю два файли: один містить код основної програми (тобто функції, що реалізовують облік речей ломбарду), а інший – методи тестування кожної з цих функцій.

Структура даних для зберігання елементів – словник, в якому ключ – це назва предмету (string), а значення – його ціна (int/float).

Файл основної програми має вигляд:

```python
def add_good(goods, name, price):
    if isinstance(name, str) and isinstance(price, (int, float)):
        goods[name] = price
        return True
    else:
        raise TypeError("Invalid data type")

def remove_good(goods, name):
    if name in goods:
        del goods[name]
        return goods
    else:
        raise ValueError(f"The good '{name}' is not found")

def get_goods_amount(goods):
    return len(goods)

def get_total_cost(goods):
    total_cost = sum(goods.values())
    return total_cost
```

- add_good – додає новий предмет, при цьому перевіряє формат введених даних, щоб назва предмету була строго рядком, а ціна – числом. Інакше виникає помилка TypeError, яка сповіщає про неправильний формат даних, внаслідок чого такий предмет відповідно не може бути додано.
- remove_good – видаляє предмет за ключем (тобто за його назвою). Якщо предмета з такою назвою не існує або його вже було видалено раніше, то виникає помилка ValueError, яка повідомляє, що предмет з таким іменем не знайдено.
- get_goods_amount – дозволяє отримати поточну кількість речей у «ломбарді».
- get_total_cost – повертає загальну вартість усіх речей, які нині зберігаються в «ломбарді».

Розгляньмо файл з тестами:

```python
import unittest
import pawnshop as ps

class TestGoodsFunctions(unittest.TestCase):
    def setUp(self):
        self.goods = {"Radio": 200, "Headphones": 300}

    def test_add_good_normal(self):
        ps.add_good(self.goods, "Laptop", 1000)
        self.assertIn("Laptop", self.goods)

    @unittest.expectedFailure
    def test_add_good_wrong(self):
        self.assertTrue(ps.add_good(self.goods, 700, "Necklace"))
        self.assertTrue(ps.add_good(self.goods, "Phone", "900"))
        self.assertTrue(ps.add_good(self.goods, 500, 400))

    def test_remove_good(self):
        ps.remove_good(self.goods, "Radio") #removing radio and checking that it doesn't exist anymore
        self.assertNotIn("Radio", self.goods)

        with self.assertRaises(ValueError): #trying to remove radio again and facing error
            ps.remove_good(self.goods, "Radio")

    def test_get_goods_amount(self):
        amount = ps.get_goods_amount(self.goods)
        self.assertEqual(amount, 2)

    def test_get_total_cost(self):
        total_cost = ps.get_total_cost(self.goods)
        self.assertEqual(total_cost, 500)
```

- setUp – метод, що автоматично викликається перед кожним тестовим методом. В цьому коді його суть полягає в ініціалізації словника, дані якого буде використано для тестування. Це необхідно для того, щоб усі тестові методи працювали в однакових умовах і в разі чого можна було легко визначити, які саме тести провалюються.

- test_add_good_normal – тестує функцію add_good за умови правильного формату введених даних, і перевіряє наявність даного предмету у словнику.
- test_add_good_wrong – так само тестує функцію add_good, проте, на відміну від попередньої функції, робить це з неправильними наборами вхідних даних. Декоратор @unittest.expectedFailure використовується для уникнення повідомлень про помилки при виконанні тесту. Таким чином, ми ніби «передбачаємо», що цей тест буде провалено і вказуємо інтерпретатору, щоб він сприйняв помилку, як очікувану і не відображав її у вікні терміналу.

Порівняємо результати тестування з використанням декоратора і без:

Без:

```
.E...
======================================================================
ERROR: test_add_good_wrong (__main__.TestGoodsFunctions.test_add_good_wrong)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "d:\KNU_FIT\3rd_grade\Programming technologies\pt2023\Lab2\pawnshop_tests.py", line 14, in test_add_good_wrong
    self.assertTrue(ps.add_good(self.goods, 700, "Necklace"))
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "d:\KNU_FIT\3rd_grade\Programming technologies\pt2023\Lab2\pawnshop.py", line 6, in add_good
    raise TypeError("Invalid data type")
TypeError: Invalid data type

----------------------------------------------------------------------
Ran 5 tests in 0.003s
```

З:

```
.x...
----------------------------------------------------------------------
Ran 5 tests in 0.003s

OK (expected failures=1)
```

- test_remove_good – тестує функцію видалення предмету і перевіряє, що він був дійсно видалений. Оператор with означає, що видалення неіснуючого предмету спричиняє помилку ValueError. Тут явно вказано те, що помилка повинна бути викликана. Отже, тест буде пройдено успішно, тому використання декоратора @unittest.expectedFailure в цьому методі не має сенсу.
- test_get_goods_amount – тестує функцію отримання поточної кількості предметів.
- test_get_total_cost – тестує функцію отримання загальної вартості предметів.

Висновок: в ході виконання лабораторної роботи ми розглянули принципи модульного тестування програмного забезпечення та на практиці засвоїли створення модульних тестів з використанням Python-фреймворку unittest.