МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»  
(Самарский университет)

Институт информатики и кибернетики\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра программных систем\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**ОТЧЕТ**  
  
по лабораторной работе №3

«Модульное тестирование»

по дисциплине «Тестирование и отладка ПО»

Обучающаяся в группе 6301-020302D Соколова Алёна

Преподаватель Лобанков Антон Алексеевич

Самара 2024

**Оглавление**

[Реализация unit-тестов 2](#_Toc181021639)

[Заключение 8](#_Toc181021640)

[Приложение 9](#_Toc181021641)

**Вариант:** 26 – Генератор случайных чисел.

**Цель работы**

Целью выполнения данной лабораторной работы является ознакомление с процессом создания unit-тестов для проверки корректности работы отдельных модулей программного обеспечения. В ходе работы необходимо изучить принципы написания unit-тестов, создать и протестировать набор тестов для программы, соответствующей индивидуальному варианту задания.

**Задание**

1. Реализовать unit-тесты для проверки работоспособности приложения (в соответствии с индивидуальным задание из л/р №1). Необходимо по максимуму протестировать тест-кейсы из л/р №2;
2. Составить отчет о проделанной работе.

# Реализация unit-тестов

В данной лабораторной работе реализуется консольное приложение для генерации случайных чисел на основе последовательности Фибоначчи с использованием языка программирования Python. Приложение включает три файла: основной файл, класс генератора случайных чисел и файл с юнит-тестами. Каждый из этих файлов играет свою роль в общем процессе работы программы. Для автоматизированного тестирования кода будет использоваться модуль unittest, что обеспечит удобство проверки корректности работы приложения и выявление возможных ошибок.

В файле FibonacciRandomGenerator.py определён класс FibonacciRandomGenerator, который отвечает за генерацию случайных чисел. Класс инициализируется с минимальным значением, максимальным значением и количеством чисел, которые необходимо сгенерировать.

class FibonacciRandomGenerator:  
def \_\_init\_\_(self, min\_value, max\_value, count):  
 if not isinstance(min\_value, (int, float)) or not isinstance(max\_value, (int, float)):  
if not isinstance(min\_value, (int, float)) or not isinstance(max\_value, (int, float)):  
 raise ValueError("Ошибка: введите числовые значения для диапазона")  
if not isinstance(count, int):  
 raise ValueError("Ошибка: введите числовое значение для количества чисел")  
if count <= 0:  
 raise ValueError("Ошибка: количество чисел должно быть положительным")  
if count > 1000000:  
 raise ValueError("Ошибка: превышено допустимое количество чисел (до 1 000 000)")  
if min\_value > max\_value:  
 raise ValueError("Ошибка: минимальное значение не может быть больше максимального")

self.min\_value = min\_value  
 self.max\_value = max\_value  
 self.count = count

Метод generate отвечает за саму логику генерации. В этом методе сначала создаётся последовательность Фибоначчи нужной длины. Для этого вызывается статический метод fibonacci, который возвращает список чисел Фибоначчи.

def generate(self):fib\_numbers = self.fibonacci(self.count + 2) *# +2 для получения нужного количества* random\_numbers = [  
 (fib\_numbers[i] % (self.max\_value - self.min\_value + 1)) + self.min\_value   
 for i in range(2, self.count + 2)  
 ]  
 return random\_numbers

В этой части кода производится модификация чисел Фибоначчи с использованием остатка от деления для получения случайных чисел в заданном диапазоне. Каждое число в диапазоне определяется как остаток от деления числа Фибоначчи на длину диапазона с последующим добавлением минимального значения.

Метод fibonacci реализует саму логику построения последовательности Фибоначчи. Он создаёт список чисел Фибоначчи до заданного числа n.

@staticmethod  
def fibonacci(n):  
 a, b = 0, 1  
 fib\_sequence = []  
 for \_ in range(n):  
 fib\_sequence.append(a)  
 a, b = b, a + b  
 return fib\_sequence

Файл Testfibonaccirandomgenerator.py содержит тесты для класса fibonaccirandomgenerator. Используется модуль unittest для организации тестов. Каждый тест проверяет определённые аспекты работы генератора, включая успешную генерацию чисел, обработку ошибок при вводе неверных значений и проверку на корректность работы с границами диапазона.

Тесты для приложения по генерации случайных чисел проверяют различные аспекты работы программы, чтобы гарантировать корректность ее функционирования.

Данный тест проверяет, что программа корректно генерирует заданное количество случайных чисел в указанном диапазоне. Например, в коде это реализовано следующим образом:

def test\_successful\_generation(self):  
 min\_value = 1  
 max\_value = 10  
 count = 5  
 print(f"Тест: Генерация чисел. Диапазон: [{min\_value}, {max\_value}], Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertEqual(len(random\_numbers), count)  
 for num in random\_numbers:  
 self.assertGreaterEqual(num, min\_value)  
 self.assertLessEqual(num, max\_value)

Этот тест создает объект генератора с диапазоном от 1 до 10 и запрашивает генерацию 5 чисел. Затем проверяется, что количество сгенерированных чисел равно 5, и каждое число находится в пределах заданного диапазона.

Тест на случай, когда минимальное значение диапазона больше максимального, проверяет, что программа правильно обрабатывает эту ошибочную ситуацию:

def test\_min\_greater\_than\_max(self):  
 print("Тест: Минимальное значение больше максимального")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator(10, 5, 5)  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: минимальное значение не может быть больше максимального")

Здесь ожидается, что при создании генератора с минимальным значением 10 и максимальным 1 будет выброшено исключение с соответствующим сообщением.

Тест на обработку нечисловых значений проверяет, что программа корректно реагирует на неверный ввод:

def test\_invalid\_input\_range(self):  
 print("Тест: Нечисловое значение в диапазоне")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator("abc", 10, 5)  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: введите числовые значения для диапазона")

Если пользователь введет строку вместо числа, программа должна выбросить исключение с сообщением об ошибке.

Тест на ввод нечислового значения для количества чисел аналогично проверяет корректность обработки ввода:

def test\_non\_numeric\_count(self):  
 print("Тест: Нечисловое значение для количества чисел")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator(1, 10, "abc")  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: введите числовое значение для количества чисел")

Если ввести строку вместо числа, программа также должна вывести сообщение об ошибке.

Тест на генерацию слишком большого количества чисел проверяет, что программа ограничивает количество генерируемых значений до 1 000 000:

def test\_large\_number\_generation(self):  
 print("Тест: Количество чисел больше допустимого")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator(1, 10, 1000001)  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: превышено допустимое количество чисел (до 1 000 000)")

В этом случае, если пользователь введет количество чисел больше 1 000 000, ожидается соответствующее сообщение об ошибке.

Тест на генерацию случайных чисел в диапазоне отрицательных значений проверяет, что программа правильно генерирует числа в заданном отрицательном диапазоне:

def test\_negative\_range(self):  
 min\_value = -10  
 max\_value = -1  
 count = 5  
 print(f"Тест: Генерация чисел в отрицательном диапазоне. Диапазон: [{min\_value}, {max\_value}], Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertTrue(all(min\_value <= num <= max\_value for num in random\_numbers))

Этот тест проверяет, что все сгенерированные числа находятся в пределах от -10 до -1.

Тест на обработку вещественных чисел можно реализовать, если программа будет поддерживать генерацию чисел с плавающей точкой:

def test\_floating\_point\_range(self):  
 min\_value = 1.1  
 max\_value = 10.5  
 count = 5  
 print(f"Тест: Генерация чисел с плавающей запятой. Диапазон: [{min\_value}, {max\_value}], Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertTrue(all(min\_value <= num <= max\_value for num in random\_numbers))

Этот тест проверяет, что программа правильно работает с вещественными числами и генерирует их в заданном диапазоне.

Тест на генерацию чисел с одинаковыми границами диапазона проверяет, что программа правильно обрабатывает случай, когда минимальное и максимальное значение совпадают:

def test\_generation\_with\_equal\_bounds(self):  
 min\_value = 5  
 max\_value = 5  
 count = 3  
 print(f"Тест: Генерация чисел при равных границах. Значение: {min\_value}, Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertTrue(all(num == min\_value for num in random\_numbers))  
 self.assertEqual(len(random\_numbers), count)

В этом случае программа должна вернуть три числа, каждое из которых равно 5.

Эти тесты обеспечивают комплексную проверку функциональности приложения, позволяя выявить и устранить возможные ошибки и недочеты в его работе. На рисунке 1 отображен результат проверки тестов.

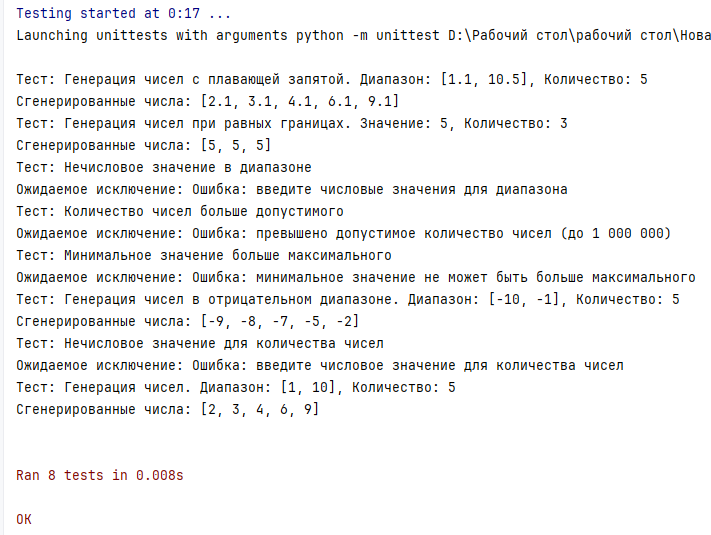


Рисунок 1 – Результаты тестов

# Заключение

В ходе выполнения работы были изучены принципы написания unit-тестов, что позволило глубже понять важность тестирования в разработке программного обеспечения. Были разработаны и выполнены unit-тесты для приложения, предназначенного для генерации случайных чисел на основе последовательности Фибоначчи. Тесты охватывают ключевые функциональные аспекты программы, включая обработку различных входных данных и генерацию чисел в заданных диапазонах. Проведенные тестирования подтвердили корректность работы основных функций приложения, что является важным шагом для обеспечения его надежности и стабильности.

# Приложение

Файл FibonacciRandomGenerator.py:

class FibonacciRandomGenerator:  
 def \_\_init\_\_(self, min\_value, max\_value, count):  
 if not isinstance(min\_value, (int, float)) or not isinstance(max\_value, (int, float)):  
 raise ValueError("Ошибка: введите числовые значения для диапазона")  
 if not isinstance(count, int):  
 raise ValueError("Ошибка: введите числовое значение для количества чисел")  
 if count <= 0:  
 raise ValueError("Ошибка: количество чисел должно быть положительным")  
 if count > 1000000:  
 raise ValueError("Ошибка: превышено допустимое количество чисел (до 1 000 000)")  
 if min\_value > max\_value:  
 raise ValueError("Ошибка: минимальное значение не может быть больше максимального")  
  
 self.min\_value = min\_value  
 self.max\_value = max\_value  
 self.count = count  
  
 def generate(self):  
 *# Генерация случайных чисел на основе последовательности Фибоначчи* fib\_numbers = self.fibonacci(self.count + 2) *# +2 для получения нужного количества* random\_numbers = [  
 (fib\_numbers[i] % (self.max\_value - self.min\_value + 1)) + self.min\_value   
 for i in range(2, self.count + 2)  
 ]  
 return random\_numbers  
  
 @staticmethod  
 def fibonacci(n):  
 a, b = 0, 1  
 fib\_sequence = []  
 for \_ in range(n):  
 fib\_sequence.append(a)  
 a, b = b, a + b  
 return fib\_sequence

Файл TestFibonacciRandomGenerator.py:

import unittest  
from FibonacciRandomGenerator import FibonacciRandomGenerator  
  
  
class TestFibonacciRandomGenerator(unittest.TestCase):  
  
 def test\_floating\_point\_range(self):  
 min\_value = 1.1  
 max\_value = 10.5  
 count = 5  
 print(f"Тест: Генерация чисел с плавающей запятой. Диапазон: [{min\_value}, {max\_value}], Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertTrue(all(min\_value <= num <= max\_value for num in random\_numbers))  
  
 def test\_generation\_with\_equal\_bounds(self):  
 min\_value = 5  
 max\_value = 5  
 count = 3  
 print(f"Тест: Генерация чисел при равных границах. Значение: {min\_value}, Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertTrue(all(num == min\_value for num in random\_numbers))  
 self.assertEqual(len(random\_numbers), count)  
  
 def test\_invalid\_input\_range(self):  
 print("Тест: Нечисловое значение в диапазоне")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator("abc", 10, 5)  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: введите числовые значения для диапазона")  
  
 def test\_large\_number\_generation(self):  
 print("Тест: Количество чисел больше допустимого")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator(1, 10, 1000001)  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: превышено допустимое количество чисел (до 1 000 000)")  
  
 def test\_min\_greater\_than\_max(self):  
 print("Тест: Минимальное значение больше максимального")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator(10, 5, 5)  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: минимальное значение не может быть больше максимального")  
  
 def test\_negative\_range(self):  
 min\_value = -10  
 max\_value = -1  
 count = 5  
 print(f"Тест: Генерация чисел в отрицательном диапазоне. Диапазон: [{min\_value}, {max\_value}], Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertTrue(all(min\_value <= num <= max\_value for num in random\_numbers))  
  
 def test\_non\_numeric\_count(self):  
 print("Тест: Нечисловое значение для количества чисел")  
 with self.assertRaises(ValueError) as context:  
 FibonacciRandomGenerator(1, 10, "abc")  
 print(f"Ожидаемое исключение: {str(context.exception)}")  
 self.assertEqual(str(context.exception), "Ошибка: введите числовое значение для количества чисел")  
  
 def test\_successful\_generation(self):  
 min\_value = 1  
 max\_value = 10  
 count = 5  
 print(f"Тест: Генерация чисел. Диапазон: [{min\_value}, {max\_value}], Количество: {count}")  
 generator = FibonacciRandomGenerator(min\_value, max\_value, count)  
 random\_numbers = generator.generate()  
 print(f"Сгенерированные числа: {random\_numbers}")  
 self.assertEqual(len(random\_numbers), count)  
 for num in random\_numbers:  
 self.assertGreaterEqual(num, min\_value)  
 self.assertLessEqual(num, max\_value)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()