|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| *Федеральное государственное бюджетное образовательное учреждение высшего образования* ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** |

Институт Информационных технологий (ИТ)

Кафедра Математического обеспечения и стандартизации информационных технологий (МОСИТ)

| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 2** |
| --- |
| **по дисциплине** |
| **«Тестирование и верификация программного обеспечения»**  **Тема: «МОДУЛЬНОЕ И МУТАЦИОННОЕ ТЕСТИРОВАНИЕ**  **ПРОГРАММНОГО ПРОДУКТА»** |

| Выполнил студент группы ИКБО-42-23 | \_Голев С.С\_ |
| --- | --- |
| Принял | Чернов Е.А |

| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_2025 г. | *(подпись студента)* |
| --- | --- | --- |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_2025 г. | *(подпись руководителя)* |

# СОДЕРЖАНИЕ

[ЦЕЛЬ И ЗАДАЧИ РАБОТЫ 3](#_xv79yema426e)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 4](#_7kk6fzk5rgdl)

[ЗАКЛЮЧЕНИЕ 12](#_xpfu9n6o3dau)

# 

# ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы: познакомить студентов с процессом модульного и мутационного тестирования, включая разработку, проведение тестов, исправление ошибок, анализ тестового покрытия, а также оценку эффективности тестов путем применения методов мутационного тестирования.

Для достижения поставленной цели работы студентам необходимо выполнить ряд задач:

− изучить основы модульного тестирования и его основные принципы;

− освоить использование инструментов для модульного тестирования (pytest для Python, JUnit для Java и др.);

− разработать модульные тесты для программного продукта и проанализировать их покрытие кода;

− изучить основы мутационного тестирования и освоить инструменты для его выполнения (MutPy, PIT, Stryker);

− применить мутационное тестирование к программному продукту, оценить эффективность тестов;

− улучшить существующий набор тестов, ориентируясь на результаты мутационного тестирования;

− оформить итоговый отчёт с результатами проделанной работы.

Состав команды: Голев С.С., Кульпин Е.А., Матяшов В.В., Петров В.Ю.

# 

# 

# ПРАКТИЧЕСКАЯ ЧАСТЬ

Распишем модули и приведем их исходный код.

**validate\_email(email)**

| def validate\_email(email):  if type(email) != str:  return TypeError('String value expected')  return '@' in email and '.' in email.split('@')[-1] |
| --- |

* Проверяет, является ли email строкой.
* Проверяет наличие символа @ и хотя бы одной точки . после @.
* Возвращает True, если email корректный, False — если нет.
* Если тип не строка — возвращает TypeError.

**get\_range(lst)**

| def get\_range(lst):  if type(lst) not in [list, tuple, set]:  return TypeError('List value expected')  for i in lst:  if type(i) != int:  return TypeError('List must consist from integer values expected')  return min(lst), max(lst) |
| --- |

* Проверяет, является ли lst списком, кортежем или множеством.
* Проверяет, что все элементы — целые числа.
* Возвращает кортеж (min(lst), max(lst)).
* Если типы некорректны — возвращает TypeError.

**only\_even(lst)**

| def only\_even(lst):  if type(lst) not in [list, tuple, set]:  return TypeError('List value expected')  for i in lst:  if type(i) != int:  return TypeError('List must consist from integer values expected')  for i in lst:  if i % 2 != 0:  return False  return True |
| --- |

* Проверяет, что lst — список/кортеж/множество и состоит из целых чисел.
* Возвращает True, если все числа четные, иначе False.
* При некорректном типе элементов или списка — возвращает TypeError.

**vector\_multiplier(vec1, vec2)**

| def vector\_multiplier(vec1, vec2):  if len(vec1) != len(vec2):  return ValueError('Vectors length must be equal')  if type(vec1) not in [list, tuple, set] or type(vec2) not in [list, tuple, set]:  return TypeError('List value expected')  for i in vec1:  if type(i) != int:  return TypeError('List must consist from integer values expected')  for i in vec2:  if type(i) != int:  return TypeError('List must consist from integer values expected')  return [(vec1[i] \* vec2[i]) for i in range(len(vec1))] |
| --- |

* Проверяет, что vec1 и vec2 одного типа: список/кортеж/множество и одинаковой длины.
* Проверяет, что все элементы — целые числа.
* Возвращает список произведений элементов с одинаковыми индексами.
* При несоответствии типов или длины — возвращает TypeError или ValueError.

**upper\_case(string)**

| def upper\_case(string):  if type(string) != str:  return TypeError('String value expected')  return string.upper() |
| --- |

* Проверяет, что string — строка.
* Возвращает строку в верхнем регистре.
* При некорректном типе — возвращает TypeError.

Проведём модульное тестирование, дабы убедиться в корректной работе функций.

**test\_validate\_email()**

| def test\_validate\_email():  assert validate\_email("test@example.com") == True  assert validate\_email("a@b.c") == True  assert validate\_email("invalid.email") == False  assert validate\_email("user@") == False  assert validate\_email("@domain.com") == False  assert validate\_email("") == False  assert isinstance(validate\_email(123), TypeError) |
| --- |

Тест проверяет функцию validate\_email.

* Сначала проводится позитивное тестирование: проверяются корректные email, такие как "test@example.com" и "a@b.c", ожидается True.
* Далее — негативное тестирование: проверяются некорректные email, например "invalid.email", "user@", "@domain.com" и пустая строка, ожидается False.
* Наконец, проверяется тестирование типизации входных данных: если передан не строковый тип (123), функция должна вернуть TypeError.

Методология: комбинация позитивного и негативного тестирования с проверкой обработки некорректного типа аргумента.

**test\_get\_range()**

| def test\_get\_range():  assert get\_range([1, 2, 3, 4, 5]) == (1, 5)  assert get\_range([-5, 0, 5]) == (-5, 5)  assert get\_range([10]) == (10, 10)  assert get\_range((1, 3, 2)) == (1, 3)  assert get\_range({4, 2, 6}) == (2, 6)  assert isinstance(get\_range("string"), TypeError)  assert isinstance(get\_range([1, 2, "3"]), TypeError) |
| --- |

Тест проверяет функцию get\_range.

* Проводится позитивное тестирование с корректными коллекциями: список [1,2,3,4,5], кортеж (1,3,2) и множество {4,2,6}. Проверяется, что функция возвращает правильный диапазон (min, max).
* Для смешанных значений, таких как [-5,0,5], и одиночного элемента [10] применяется граничное тестирование, чтобы убедиться, что функция корректно обрабатывает отрицательные числа и минимальные размеры коллекций.
* Для некорректного типа входных данных, например "string", применяется тестирование типизации — ожидается TypeError.  
  Для коллекций с неверными элементами, например [1,2,"3"], используется негативное тестирование, проверяющее, что функция возвращает TypeError.

Методология: сочетание позитивного, негативного и граничного тестирования, с проверкой типов.

**test\_only\_even()**

| def test\_only\_even():  # Тест только четных чисел  assert only\_even([2, 4, 6, 8]) == True  assert only\_even([0, 2, 4]) == True  assert only\_even([-2, -4, 0]) == True  assert only\_even([1, 2, 3]) == False  assert only\_even([2, 4, 5]) == False  assert only\_even([1]) == False  assert only\_even((2, 4, 6)) == True # tuple  assert only\_even({2, 4, 8}) == True # set  assert isinstance(only\_even("string"), TypeError)  assert isinstance(only\_even([1, "2"]), TypeError) |
| --- |

Тест проверяет функцию only\_even.

* Проводится позитивное тестирование: списки [2,4,6,8], кортежи (2,4,6) и множества {2,4,8} должны возвращать True, так как все элементы четные.
* Негативное тестирование проверяет случаи, когда есть нечетные числа: [1,2,3], [2,4,5] и [1] должны возвращать False.
* Тестирование типизации входных данных проверяет, что при передаче строки "string" или коллекции с некорректными элементами [1,"2"] возвращается TypeError.

Методология: позитивное и негативное тестирование с контролем типов и поддержки разных типов коллекций.

**test\_vector\_multiplier()**

| def test\_vector\_multiplier():  assert vector\_multiplier([1, 2, 3], [4, 5, 6]) == [4, 10, 18]  assert vector\_multiplier([0, 1], [2, 3]) == [0, 3]  assert vector\_multiplier([-1, 2], [3, -4]) == [-3, -8]  assert vector\_multiplier((1, 2), (3, 4)) == [3, 8]  assert isinstance(vector\_multiplier([1, 2], [1, 2, 3]), ValueError)  assert isinstance(vector\_multiplier("string", [1, 2]), TypeError)  assert isinstance(vector\_multiplier([1, "2"], [1, 2]), TypeError)  assert isinstance(vector\_multiplier([1, 2], [1, "2"]), TypeError) |
| --- |

Тест проверяет функцию vector\_multiplier.

* Позитивное тестирование выполняется для корректных векторов одинаковой длины: [1,2,3] \* [4,5,6] и (1,2) \* (3,4) должны возвращать правильный результат произведений.
* Граничное тестирование проверяет нули и отрицательные числа: [0,1] \* [2,3] и [-1,2] \* [3,-4].
* Негативное тестирование проверяет несовпадение длины векторов: [1,2] \* [1,2,3] должно возвращать ValueError.
* Тестирование типизации входных данных проверяет, что строки или другие некорректные типы коллекций ("string", [1,2]) и элементы [1,"2"] вызывают TypeError.

Методология: сочетание позитивного, негативного и граничного тестирования с контролем типов элементов и длины коллекций.

**test\_upper\_case()**

| def test\_upper\_case():  assert upper\_case("hello") == "HELLO"  assert upper\_case("Python") == "PYTHON"  assert upper\_case("test case") == "TEST CASE"  assert upper\_case("") == ""  assert upper\_case("123abc") == "123ABC" |
| --- |

Тест проверяет функцию upper\_case.

* Позитивное тестирование проверяет, что строки "hello", "Python", "test case", пустая строка "" и строка с цифрами "123abc" корректно преобразуются в верхний регистр.
* Тестирование типизации входных данных проверяет, что некорректные типы (123, []) вызывают TypeError.

Методология: проверка корректности работы функции на разных типах строк и проверка типов входных данных.

Проведём первый запуск тестов:

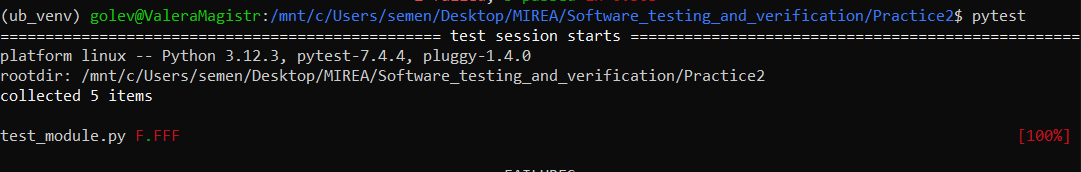
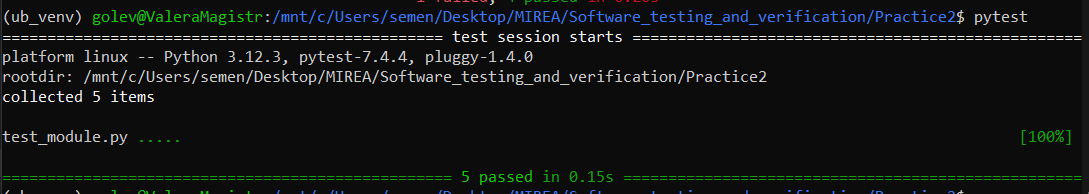


Рисунок 1 – Первый запуск теста с ошибками

Рисунок 2 – Первый запуск теста после исправлений

Проведём мутационное тестирование на основе мутационной системы тестирования для Python Mutmut.

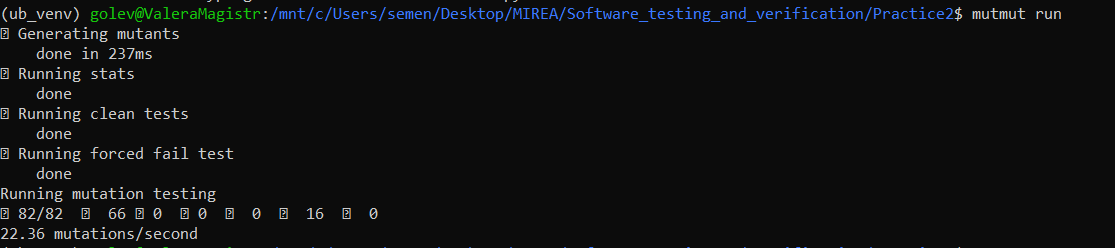


Рисунок 3 – Первый запуск мутационного тестирования

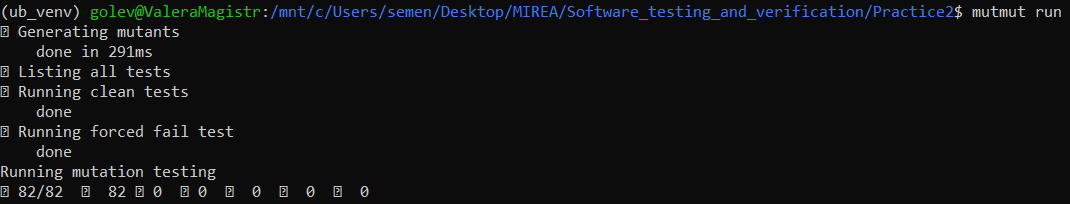


Рисунок 4 –Запуск мутационного тестирования после добавления новых тестов

# ЗАКЛЮЧЕНИЕ

Анализ проведенного тестирования показывает, что набор тестов обеспечивает достаточно высокое покрытие функций как по позитивным, так и по негативным сценариям. Тесты проверяют корректность работы функций с правильными данными, обработку ошибок типов, а также граничные случаи, такие как пустые коллекции или одиночные элементы. Это позволяет с высокой вероятностью выявлять ошибки логики и некорректную работу функций.

Итоговые выводы показывают, что текущее тестирование обеспечивает базовую уверенность в корректности функций, однако для повышения надежности системы необходимо дополнить тесты сценариями, выявляющими логические ошибки, которые могут быть пропущены стандартными проверками. Мутационное тестирование продемонстрировало эффективность текущих тестов и указало на направления для их улучшения, что позволяет сделать функции более устойчивыми к ошибкам и повысить качество кода.