CS 520
In-class exercise 2
Software testing

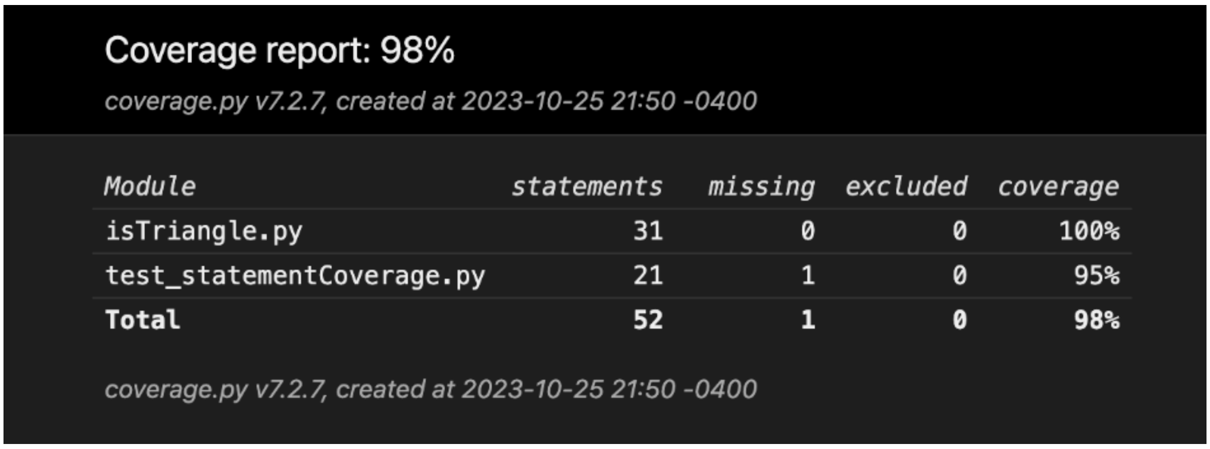
Team: **CodeCraftersH**
Git: https://github.com/HEMANGANI/IE2-Triangle
Members: **Harshita Loganathan, Hemangani Nagarajan**

Control Flow Graph:

```
1.  Find the execution flow for two (2) other normative cases from the CFG file:
equilateral triangle, and isosceles triangle
a. Equilateral Triangle: 1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 14 -> 15
b. Isosceles Triangle: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9 -> 10 -> 14 -> 16 -> 17
2. Find the execution flow for two (2) exceptional cases from the CFG file: invalid
sides and violates triangle inequality
a. Invalid Sides: 1 -> 2
b. Violates Triangle Inequality: 1 -> 3 -> 4 -> 6 -> 8 -> 10 -> 11 -> 12
```

Testing & Analysis:

Statement Coverage:

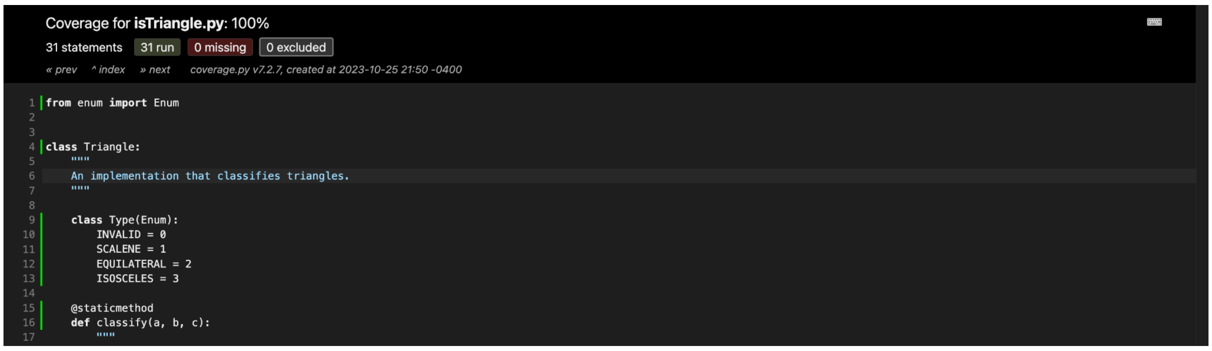


Coverage report: 98%

coverage.py v7.2.7, created at 2023-10-25 21:50 -0400

Module	statements	missing	excluded	coverage
isTriangle.py	31	0	0	100%
test_statementCoverage.py	21	1	0	95%
Total	**52**	**1**	**0**	**98%**

coverage.py v7.2.7, created at 2023-10-25 21:50 -0400



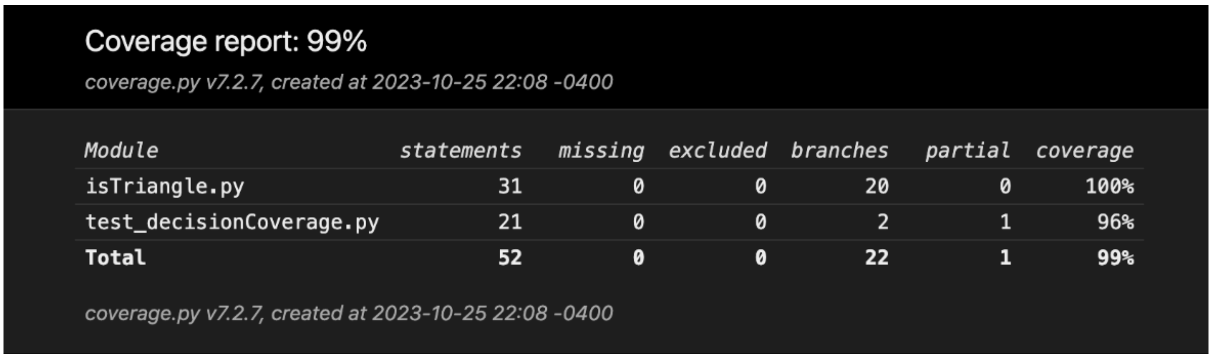
Coverage for **isTriangle.py**: 100%

31 statements 31 run 0 missing 0 excluded

« prev ^ index » next coverage.py v7.2.7, created at 2023-10-25 21:50 -0400

```
from enum import Enum


class Triangle:
    """
    An implementation that classifies triangles.
    """

    class Type(Enum):
        INVALID = 0
        SCALENE = 1
        EQUILATERAL = 2
        ISOSCELES = 3

    @staticmethod
    def classify(a, b, c):
        """
```

Decision Coverage:

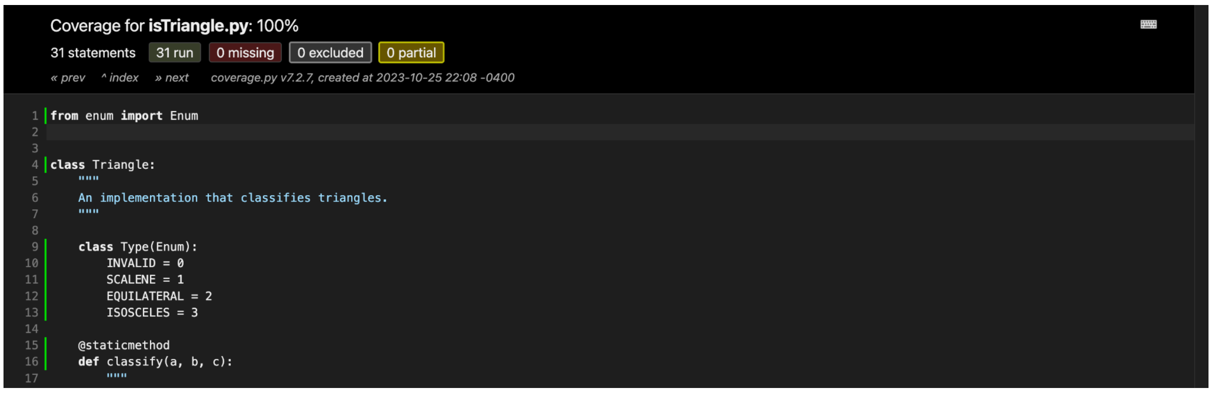


Coverage report: 99%

coverage.py v7.2.7, created at 2023-10-25 22:08 -0400

Module	statements	missing	excluded	branches	partial	coverage
isTriangle.py	31	0	0	20	0	100%
test_decisionCoverage.py	21	0	0	2	1	96%
Total	**52**	**0**	**0**	**22**	**1**	**99%**

coverage.py v7.2.7, created at 2023-10-25 22:08 -0400



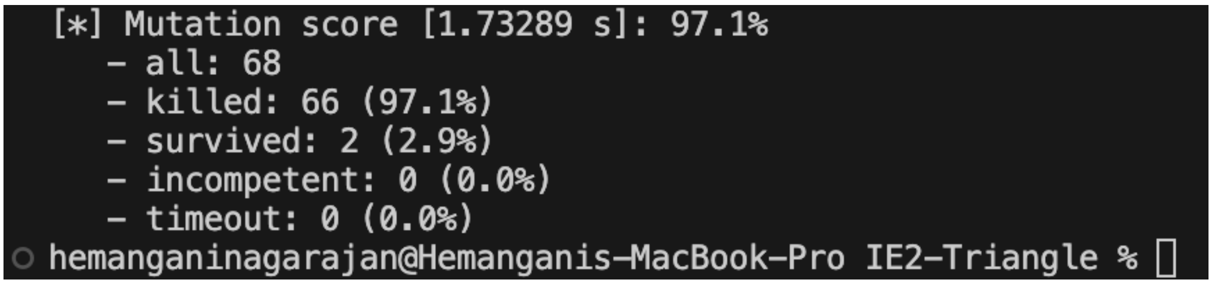
Coverage for **isTriangle.py**: 100%

31 statements 31 run 0 missing 0 excluded 0 partial

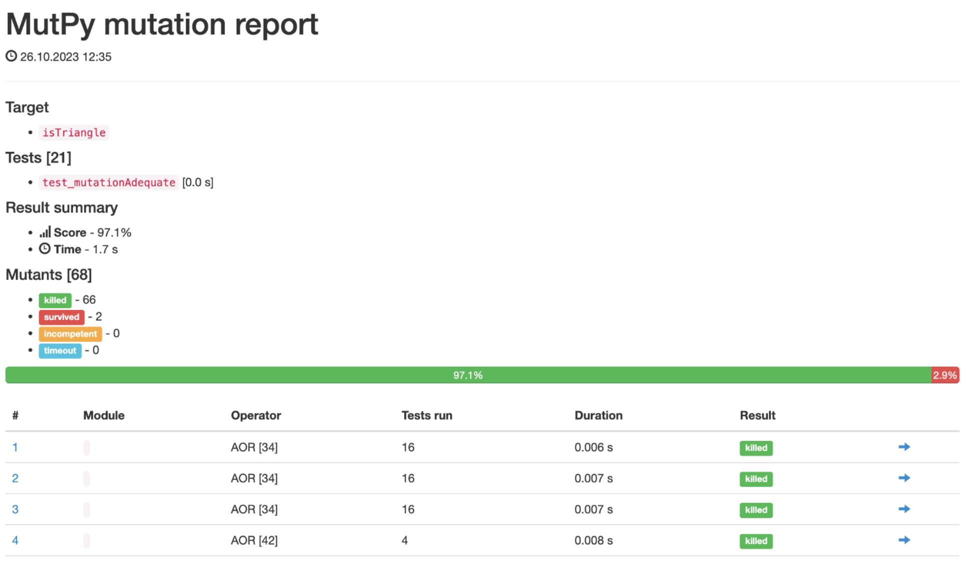
« prev ^ index » next *coverage.py v7.2.7, created at 2023-10-25 22:08 -0400*

```
from enum import Enum


class Triangle:
    """
    An implementation that classifies triangles.
    """

    class Type(Enum):
        INVALID = 0
        SCALENE = 1
        EQUILATERAL = 2
        ISOSCELES = 3

    @staticmethod
    def classify(a, b, c):
        """
```

Mutation Testing:



```
[*] Mutation score [1.73289 s]: 97.1%
   - all: 68
   - killed: 66 (97.1%)
   - survived: 2 (2.9%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
○ hemanganinagarajan@Hemanganis-MacBook-Pro IE2-Triangle %
```



MutPy mutation report

26.10.2023 12:35

Target

- isTriangle

Tests [21]

- test_mutationAdequate [0.0 s]

Result summary

- Score - 97.1%
- Time - 1.7 s

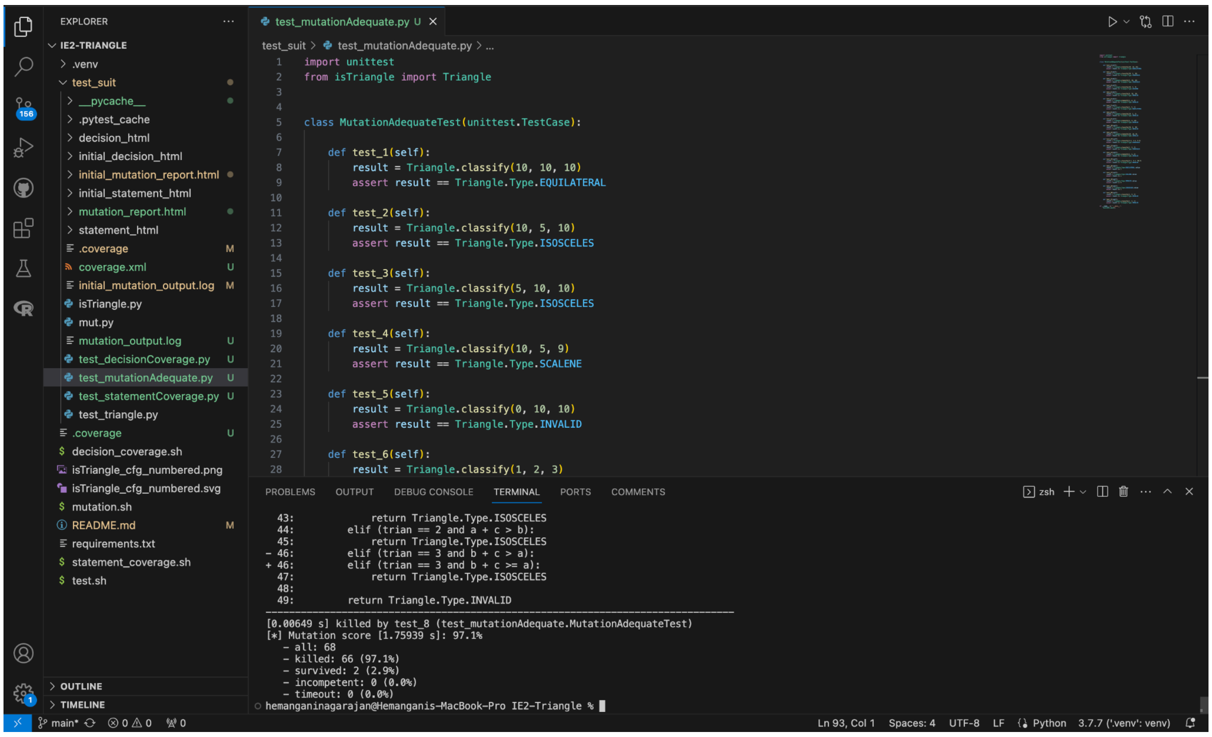
Mutants [68]

- killed - 66
- survived - 2
- incompetent - 0
- timeout - 0

97.1% 2.9%

#	Module	Operator	Tests run	Duration	Result	
1		AOR [34]	16	0.006 s	killed	➡
2		AOR [34]	16	0.007 s	killed	➡
3		AOR [34]	16	0.007 s	killed	➡
4		AOR [42]	4	0.008 s	killed	➡

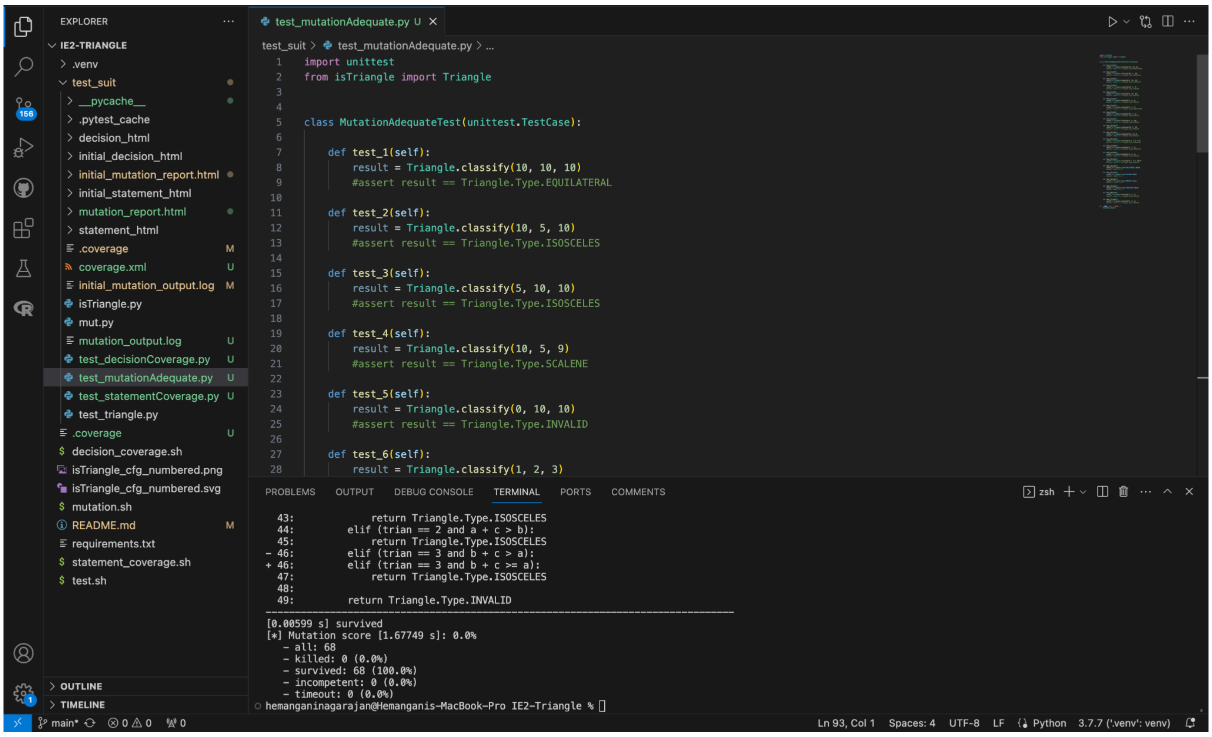
With assertions:



```python
import unittest
from isTriangle import Triangle


class MutationAdequateTest(unittest.TestCase):

    def test_1(self):
        result = Triangle.classify(10, 10, 10)
        assert result == Triangle.Type.EQUILATERAL

    def test_2(self):
        result = Triangle.classify(10, 5, 10)
        assert result == Triangle.Type.ISOSCELES

    def test_3(self):
        result = Triangle.classify(5, 10, 10)
        assert result == Triangle.Type.ISOSCELES

    def test_4(self):
        result = Triangle.classify(10, 5, 9)
        assert result == Triangle.Type.SCALENE

    def test_5(self):
        result = Triangle.classify(0, 10, 10)
        assert result == Triangle.Type.INVALID

    def test_6(self):
        result = Triangle.classify(1, 2, 3)
```

```
  43:            return Triangle.Type.ISOSCELES
  44:        elif (trian == 2 and a + c > b):
  45:            return Triangle.Type.ISOSCELES
- 46:        elif (trian == 3 and b + c > a):
+ 46:        elif (trian == 3 and b + c >= a):
  47:            return Triangle.Type.ISOSCELES
  48:
  49:        return Triangle.Type.INVALID
--------------------------------------------------------------------------------
[0.00649 s] killed by test_8 (test_mutationAdequate.MutationAdequateTest)
[*] Mutation score [1.75939 s]: 97.1%
   - all: 68
   - killed: 66 (97.1%)
   - survived: 2 (2.9%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
○ hemanganinagarajan@Hemanganis-MacBook-Pro IE2-Triangle % ▌
```

Without assertions:



```python
import unittest
from isTriangle import Triangle


class MutationAdequateTest(unittest.TestCase):

    def test_1(self):
        result = Triangle.classify(10, 10, 10)
        #assert result == Triangle.Type.EQUILATERAL

    def test_2(self):
        result = Triangle.classify(10, 5, 10)
        #assert result == Triangle.Type.ISOSCELES

    def test_3(self):
        result = Triangle.classify(5, 10, 10)
        #assert result == Triangle.Type.ISOSCELES

    def test_4(self):
        result = Triangle.classify(10, 5, 9)
        #assert result == Triangle.Type.SCALENE

    def test_5(self):
        result = Triangle.classify(0, 10, 10)
        #assert result == Triangle.Type.INVALID

    def test_6(self):
        result = Triangle.classify(1, 2, 3)
```

```
  43:            return Triangle.Type.ISOSCELES
  44:        elif (trian == 2 and a + c > b):
  45:            return Triangle.Type.ISOSCELES
- 46:        elif (trian == 3 and b + c > a):
+ 46:        elif (trian == 3 and b + c >= a):
  47:            return Triangle.Type.ISOSCELES
  48:
  49:        return Triangle.Type.INVALID
--------------------------------------------------------------------------------
[0.00599 s] survived
[*] Mutation score [1.67749 s]: 0.0%
   - all: 68
   - killed: 0 (0.0%)
   - survived: 68 (100.0%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
○ hemanganinagarajan@Hemanganis-MacBook-Pro IE2-Triangle % ▯
```

1. What are 2 best practices satisfied by the triangle project that make it easier to write the unit tests and run them?

SOLUTION:

Two recommended practices that help with unit test writing and execution are satisfied by the triangle project:

1. Relating the different triangle kinds to enumerated data types (Equilateral, Isosceles, Scalene). This means that instead of depending just on magic strings like "Equilateral," "Isosceles," etc., the test cases for the three distinct types of triangles can be easily constructed and tested. Furthermore, magic string coding is a bad practice. By putting the invalid type in the enum variable, the code makes it easy to test for invalid or unusual situations, as when the length of a side receives zero or negative values as input, or when the third side is not more than the sum of the other two sides.

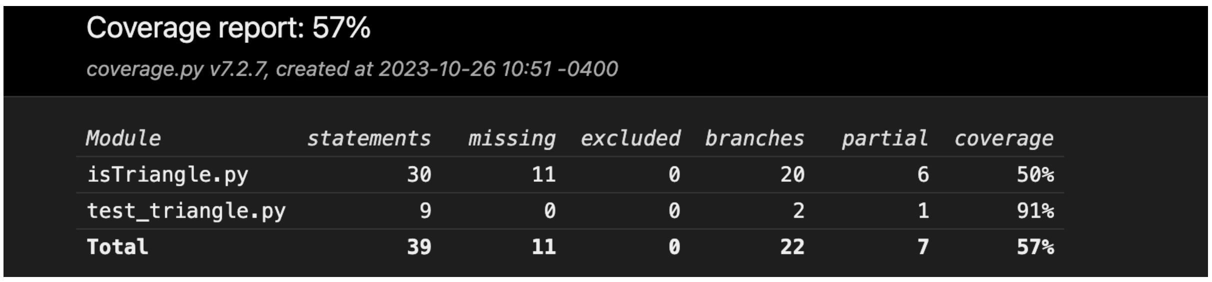
2. A README.md file is included to help with testing, and comments and documentation make the code easy to read and understand.

2. For the isTriangle class with the initial test suite, what is the statement (a.k.a. line) coverage percentage? the decision (a.k.a. branch) coverage percentage? the mutant detection rate?

SOLUTION:

For the initial test suite:

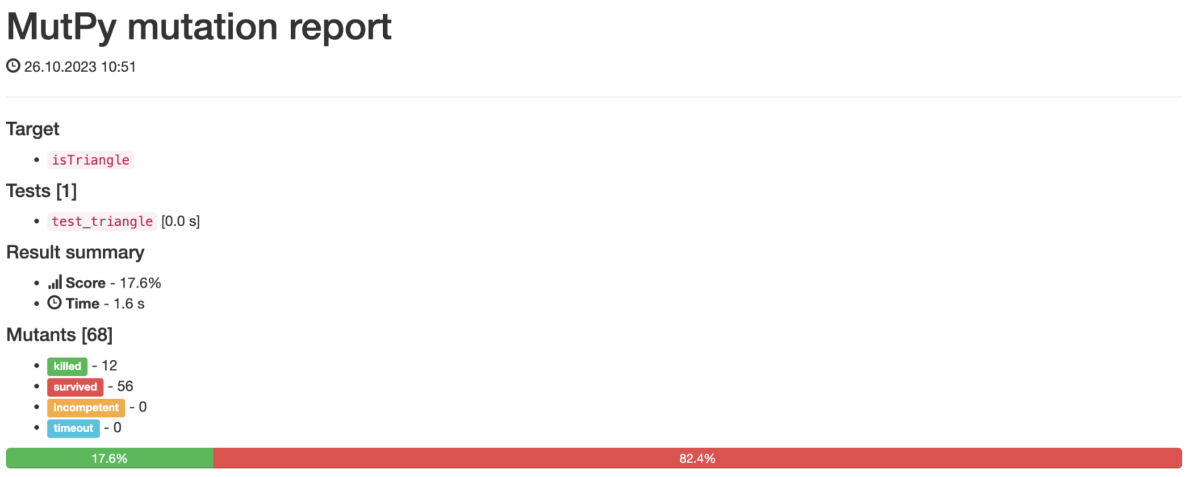
1. Statement/Line coverage percentage: 50%
2. Condition/Branch coverage percentage: 50%
3. Mutant detection rate = 17.6%



Coverage report: 57%

coverage.py v7.2.7, created at 2023-10-26 10:51 -0400

Module	statements	missing	excluded	branches	partial	coverage
isTriangle.py	30	11	0	20	6	50%
test_triangle.py	9	0	0	2	1	91%
Total	**39**	**11**	**0**	**22**	**7**	**57%**



MutPy mutation report

26.10.2023 10:51

Target

- isTriangle

Tests [1]

- test_triangle [0.0 s]

Result summary

- Score - 17.6%
- Time - 1.6 s

Mutants [68]

- killed - 12
- survived - 56
- incompetent - 0
- timeout - 0

3. Did your approach to writing unit tests differ between developing a coverage-adequate test suite and developing a mutation-adequate test suite? Briefly explain why or why not.

SOLUTION:

Coverage-based testing focuses on executing parts of the code, mutation-based testing focuses on the test suite's ability to detect changes or errors in the code. Both approaches are complementary: high coverage can ensure that all parts of the code are touched by tests, while mutation testing can ensure that these tests are genuinely effective at catching defects.

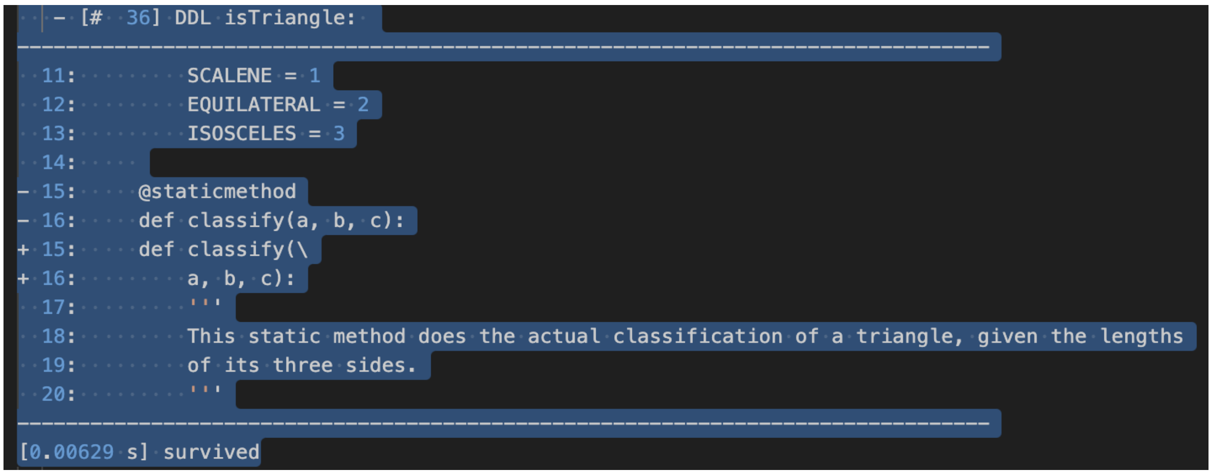
Creating a coverage-adequate test suite vs a mutation-adequate test suite requires different authoring techniques for unit tests. This is due to the fact that we ensure that each line and each condition has been evaluated while creating coverage-adequate tests. However, we search for examples where the specific mutation under consideration fails or detects the mutants in order to identify mutation-adequate tests. Therefore, 100% code coverage may or may not be present in a mutation-adequate test suite. As observed in the results above.

4. Consider your mutation-adequate test suite and the triangle program. For any given program, why are some mutants not detectable?

SOLUTION:

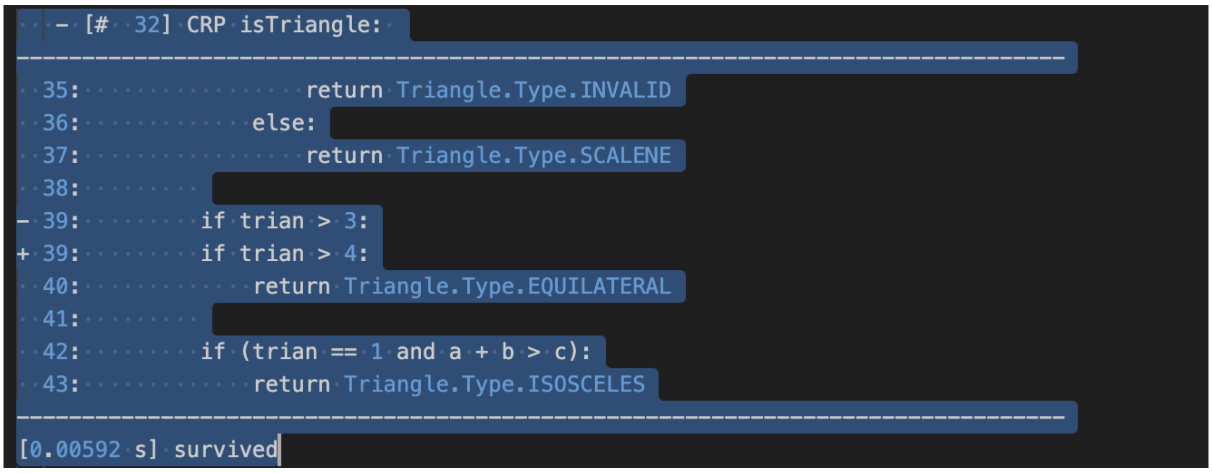
Some mutants in a program are not detectable due to various reasons, such as:
Equivalent Mutants:

These are mutants that, although different in code, end up having the same observable behavior as the original program. No matter how exhaustive your tests are, you can't kill these mutants because they don't introduce any erroneous or different behavior.



```
  - [# 36] DDL isTriangle: 
--------------------------------------------------------------------------------
  11:         SCALENE = 1
  12:         EQUILATERAL = 2
  13:         ISOSCELES = 3
  14:     
- 15:     @staticmethod
- 16:     def classify(a, b, c):
+ 15:     def classify(\
+ 16:         a, b, c):
  17:         '''
  18:         This static method does the actual classification of a triangle, given the lengths
  19:         of its three sides.
  20:         '''
--------------------------------------------------------------------------------
[0.00629 s] survived
```

Example1: The mutation removed the @staticmethod decorator from a method, yet the function can still be invoked using the class name, such as Triangle.classify(a, b, c), because Python permits calling undecorated methods via the class if they don't access instance-specific data. Additionally, a syntactical line break was added to the function declaration, which doesn't affect its behavior. Both changes don't alter the program's external behavior, making this mutation undetectable by the test suite.



```
   - [#  32] CRP isTriangle:
--------------------------------------------------------------------------------
  35:                 return Triangle.Type.INVALID
  36:             else:
  37:                 return Triangle.Type.SCALENE
  38:         
- 39:         if trian > 3:
+ 39:         if trian > 4:
  40:             return Triangle.Type.EQUILATERAL
  41:         
  42:         if (trian == 1 and a + b > c):
  43:             return Triangle.Type.ISOSCELES
--------------------------------------------------------------------------------
[0.00592 s] survived
```

Example2: The mutation changes the condition for classifying an equilateral triangle from `trian > 3` to `trian > 4`. Given the logic, an equilateral triangle would have a `trian` value of 6. Both the original and mutated conditions will classify it as equilateral. An isosceles triangle's maximum `trian` value would be 4, but no such triangle exists due to the transitive property of equality. Thus, the mutation doesn't alter the program's behavior for any possible input, making it an equivalent mutant. Equivalent mutants, by nature, can't be killed by any test case, as they don't change the program's observable behavior.

5. What changes in the code coverage percentages and mutant detection rate did you observe when deleting (or commenting out) all assertions?

SOLUTION:

After leaving all assertion statements in the test cases commented out, the code coverage percentages remain the same. The code coverage % is unaffected by the assert statements being commented, even in the absence of them, because the test cases cover every line or branch.

Eliminating the assertion statements lowers the mutation detection rate. We ensure that the code produces the right result even for inputs where altered code produces wrong results by employing assert statements. Therefore, in the absence of assert statements, it is impossible to identify the mutations, and if the assert statements are deleted or commented out, the detection rate drops to zero. The observations are pasted above.

6. Create a definition of “test case redundancy” based on code coverage or mutation analysis. Given your definition of test case redundancy, are some of the test cases in your test suites redundant? Given your definition of test case redundancy, would you remove redundant test cases? Briefly explain why or why not.

SOLUTION:

When the code produced by one test case and the code executed by another test case overlap, the test cases in software testing are deemed redundant.

Think about the scenario in which we use three different test files for mutation detection, branch coverage, and line coverage. Test cases become redundant as a result of this. To put things into perspective, let's look more closely at the code block used to identify an equilateral triangle. Redundancy results from the repetitive testing of this code logic across all three files.

However, since every line, condition, and mutation has only been evaluated once, there are no redundancies inside a single test suite, that is, if we look at each test suite independently.

Hence, by utilizing a single file for testing and eliminating such repeats, the duplication caused by using three distinct files may be eliminated. Longer testing times resulting from duplicate examples might be problematic, particularly when lengthy or sophisticated computations are involved.

In terms of testing metrics, the code coverage % and the caliber of testing needs will remain unaffected even if we only retain one copy of the redundant test case.

7. How many decision points did you find for the Control flow graph for normative cases (scalene_triangle, equilateral_triangle, and isosceles_triangle) and exception cases (invalid sides and triangle inequality)? Did these findings help you to create a better test suite?

SOLUTION:

Let's break down the decision points by analyzing the control flow:
Decision Points:

1. Check if any side is less than or equal to 0:
   if a <= 0 or b <= 0 or c <= 0:

2. Check if side `a` is equal to side `b`:
   if a == b:

3. Check if side `a` is equal to side `c`:
   if a == c:

4. Check if side `b` is equal to side `c`:
   if b == c:

5. Check if the triangle is scalene based on the value of `trian`:
   if trian == 0:

6. Inside the above condition, check for triangle inequality:
   if a + b <= c or a + c <= b or b + c <= a:

7. Check if the triangle is equilateral based on the value of `trian`:
   if trian > 3:

8, 9, and 10. Check for isosceles based on the value of `trian` and triangle inequality:

```
  if trian == 1 and a + b > c:
  elif trian == 2 and a + c > b:
  elif trian == 3 and b + c > a:
```

In analyzing the provided triangle classification code, I identified a total of 10 decision points. These decision points include conditions that check for invalid sides, equality of sides, the triangle inequality, and the specific triangle type based on the combined comparisons of the sides.

For the normative cases (scalene, equilateral, and isosceles triangles), there are a total of 7 decision points: the three checks for equality of sides (2, 3, 4), the check for scalene triangles (5), the check for equilateral triangles (7), and the three checks for isosceles triangles (8, 9, 10). For the exception cases (invalid sides and triangle inequality), there are a total of 2 decision points: the check for invalid sides (1) and the triangle inequality within the scalene condition (6).

The decision points that pertain to normative cases (scalene, equilateral, and isosceles triangles) are the ones checking for equality of sides, the value of `trian` to determine triangle type, and the triangle inequality. Those that relate to exception cases include the check for invalid sides and the triangle inequality which, if not met, classifies the triangle as invalid.

Understanding these decision points is instrumental in creating a comprehensive test suite. For a robust test suite, one would ideally have tests that cover each of these decision points, ensuring that the program works as expected in all scenarios. This granularity in testing will not only ensure the correct classification of triangle types but also validate the conditions that determine the validity of the triangle. Thus, having this detailed understanding of the control flow in the code is invaluable in achieving high coverage and reliability in the test suite.