Repo link: https://github.com/shreyabirthare/ex2compsci520

We have created 3 test suites for testing:

1. test_decisionCoverage.py
2. test_mutationAdequate.py
3. test_statementCoverage.py

The relevant contents of these can be found within the test_suit folder.

We also ran test.sh initially to see the covered lines/statements/decisions and mutant ratios. Those can be found in the test_suit directory as well.

Additionally, we also created 3 shell scripts and 3 test scripts to run and save the output and log files when we run the test suites without “assert”. Below are the names:

1. test_statementCoverageNoAssert.py
2. test_mutationAdequateNoAssert.py
3. test_decisionCoverageNoAssert.py
4. Mutation_no_assert.sh
5. Statement_coverage_no_assert.sh
6. Decision_coverage_no_assert.sh

Questions Using your notes and results, answer the following questions:

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

- Modular Code: The code is well-structured and modular. The classification logic of triangles is contained within the Triangle class. It provides a clear API for classifying triangles based on their side lengths. The code doesn't mix different concerns, making it easier to understand and test. This modularity enables writing focused unit tests for the classification logic without the need to test unrelated functionality.
- Static Method: The classification logic is implemented as a static method. This allows us to use the method without needing to create an instance of the Triangle class. It simplifies the process of testing the classification function in isolation and can call the classify method directly with different inputs in your unit tests without creating unnecessary instances of the class.
- Understandability: The code includes comments to explain complex or non-trivial sections. For instance, within the classify method, there are comments explaining the different conditions and the logic applied to classify triangles. These comments help readers understand the algorithm used for classification.

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?

Initial:

Statement Coverage: 65%

Coverage report: 47%

coverage.py v7.2.7, created at 2023-10-26 11:40 -0400

Module	*statements*	*missing*	*excluded*	*coverage*
isTriangle.py	31	11	0	65%
test_triangle.py	9	1	0	89%

Decision Coverage: 53%

Coverage report: 23%

filter...

coverage.py v7.2.7, created at 2023-10-26 11:40 -0400

Module	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
isTriangle.py	31	11	0	22	6	53%
test_triangle.py	9	0	0	2	1	91%
/opt/homebrew/lib/python3.11/site-packages/_distutils_hack/__init__.py	100	95	0	40	0	6%
Total	**140**	**106**	**0**	**64**	**7**	**23%**

Mutant Detection Rate: 23.1%

```
[*] Mutation score [1.68257 s]: 23.1%
    - all: 52
    - killed: 12 (23.1%)
    - survived: 40 (76.9%)
    - incompetent: 0 (0.0%)
    - timeout: 0 (0.0%)
```

initial_mutation_output.log

After adding test cases:

Statement Coverage: 97%

Coverage report: 48%

coverage.py v7.2.7, created at 2023-10-26 17:53 -0400

Module	*statements*	*missing*	*excluded*	*coverage*
isTriangle.py	31	1	0	97%
test_statementCoverage.py	113	4	0	96%

Decision Coverage: 100%

Coverage report: 54%

filter...

coverage.py v7.2.7, created at 2023-10-26 18:41 -0400

Module	statements	missing	excluded	branches	partial	coverage
isTriangle.py	31	0	0	22	0	100%
test_decisionCoverage.py	97	3	0	2	1	96%
/opt/homebrew/lib/python3.11/site-packages/_distutils_hack/__init__.py	100	95	0	40	0	6%
Total	**228**	**98**	**0**	**64**	**1**	**54%**

coverage.py v7.2.7, created at 2023-10-26 18:41 -0400

Mutant Detection Rate: 94.2%

```
[*] Mutation score [3.39219 s]: 94.2%
   - all: 52
   - killed: 49 (94.2%)
   - survived: 3 (5.8%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

mutation_output.log

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.
Yes, although the testing may be somewhat similar but the goal for coverage vs mutant adequate test suites are different; this required us to write test suites based on the different requirements below:

Approach for coverage adequate:

- We exercise different code paths, including edge cases and exceptional scenarios.
- Verify that the code functions correctly under various conditions.
- Test cases are designed based on the code structure and logic, aiming to touch every part of the code to increase coverage.
- Testing individual functions or methods and their different branches.
- Mutation testing may not be the primary concern, coverage metrics are essential for assessing the adequacy of the test suite.
- We ensure all lines, branches, and statements of the code are executed by the tests.

Approach for mutation adequate test suite:

- The aim is to create a test suite that can effectively identify and kill mutants, i.e., the altered versions of the code.
- So based on the different mutant variants, we had to write test cases to make sure those were robust enough to be caught incase of failures
- We changed the behavior of different variables and wrote our test cases to determine the output based on the expected and the observed behavior.

- Some approaches were reversing the inequality sign, making the method non static, adding inequalities to compare two sides of triangle, etc, all to observe the behavior of our test suite to ensure robustness

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

It is dependent on the nature of the mutation testing and our code, some of the reasons are:

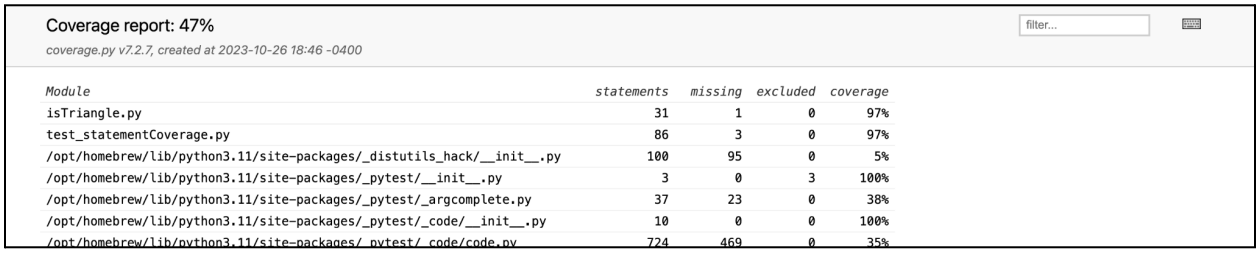
- Inadequate Test Suite: If the test suite does not sufficiently cover all possible code paths, edge cases, certain mutants might remain undetected. Lack of test coverage in specific areas of the code can result in undetected mutations, as the tests fail to execute the portions of the code affected by the mutations. In our case there are 3 misses currently as the test suite has insufficient test cases to cover all the 52 mutant variants.
- Higher-order Mutations: Mutations that involve multiple changes or complex interactions between different parts of the code can be challenging to detect and may require more extensive testing strategies to uncover the changes in program behavior. For eg when static k/w was removed we were unable to detect it with our test suite and what effect it causes.
- Complex Mutations: Certain mutations involve intricate changes that significantly alter the program's behavior, making them harder to detect using the existing test suite. Such mutations might create subtle changes that escape the detection capabilities of the current tests, especially if the tests do not cover the specific altered behavior.

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

Statement & Decision Coverage:

The coverage remains the same because the code coverage tool is measuring the percentage of lines of code that have been executed during the tests. Commenting the assertions in the coverage.py files doesn't affect the coverage percentage for the isTriangle.py since the assertions are not in the isTriangle.py file.

The screenshots below show how it is changing for both files:

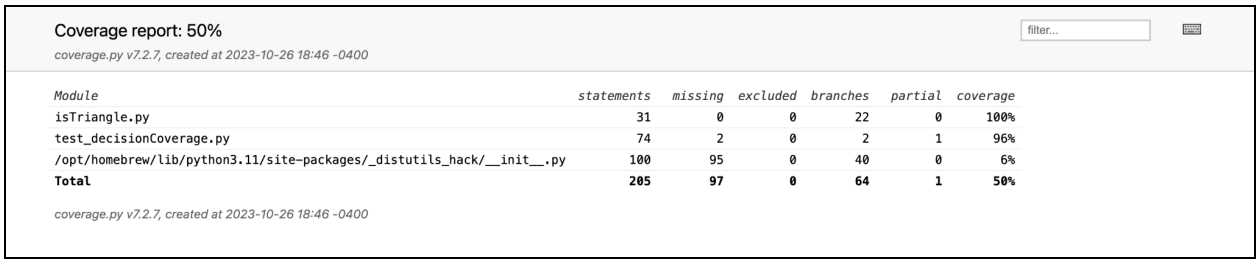


Coverage report: 47%

filter...

coverage.py v7.2.7, created at 2023-10-26 18:46 -0400

Module	statements	missing	excluded	coverage
isTriangle.py	31	1	0	97%
test_statementCoverage.py	86	3	0	97%
/opt/homebrew/lib/python3.11/site-packages/_distutils_hack/__init__.py	100	95	0	5%
/opt/homebrew/lib/python3.11/site-packages/_pytest/__init__.py	3	0	3	100%
/opt/homebrew/lib/python3.11/site-packages/_pytest/_argcomplete.py	37	23	0	38%
/opt/homebrew/lib/python3.11/site-packages/_pytest/_code/__init__.py	10	0	0	100%
/opt/homebrew/lib/python3.11/site-packages/_pytest/_code/code.py	724	469	0	35%



Coverage report: 50%

filter...

coverage.py v7.2.7, created at 2023-10-26 18:46 -0400

Module	statements	missing	excluded	branches	partial	coverage
isTriangle.py	31	0	0	22	0	100%
test_decisionCoverage.py	74	2	0	2	1	96%
/opt/homebrew/lib/python3.11/site-packages/_distutils_hack/__init__.py	100	95	0	40	0	6%
Total	**205**	**97**	**0**	**64**	**1**	**50%**

coverage.py v7.2.7, created at 2023-10-26 18:46 -0400

Mutation Coverage:

Here attaching output of two log files: mutation_output.log (has assert) and mutation_no_assert_output.log (does not have asserts). There is a significant drop in the coverage rate when we do not use asserts.

mutation_output.log:

```
1467 [*] Mutation score [3.39219 s]: 94.2%
1468    - all: 52
1469    - killed: 49 (94.2%)
1470    - survived: 3 (5.8%)
1471    - incompetent: 0 (0.0%)
1472    - timeout: 0 (0.0%)
```

mutation_no_assert_output.log

```
[*] Mutation score [3.60589 s]: 0.0%
   - all: 52
   - killed: 0 (0.0%)
   - survived: 52 (100.0%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

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.

Test case redundancy is the existence of multiple test cases within a test suite that effectively assess the same aspect of a program's behavior or cover the same code paths. Certain test cases might provide overlapping coverage, either in terms of code execution or mutation

detection or both, without introducing any new or distinct information about the program's behavior or the effectiveness of the tests. They also contribute to unnecessary repetition and might increase the testing time without providing additional insights or benefits.

Our test suites also have a bit of redundant test cases. However, in the context of removing redundant test cases, we will have to consider various factors, including the specific objectives of the testing process, the overall testing strategy, and the constraints of the development environment. If we are the ones developing various/ multiple test suites then we might be aware of the redundant cases. However in a team setting, where there are multiple contributors, different contributors might not be able to assess how a test case belonging to a test suite written by others may or may not be redundant. Here both me and my partner worked on different test suites. Hence before removing a redundant test, i will have to consult with her once to ensure that it doesn't break her coverage.

Removing redundant test cases can help streamline the testing process, reduce execution time, and improve the efficiency of the test suite. However, it is crucial to ensure that the removal of redundant test cases does not compromise the comprehensiveness of the test suite or overlook critical aspects of the program's behavior.

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?

Decision Points:

- Scalene: 6
- Equilateral: 6
- Isosceles: 7
- Invalid: 1
- Triangle Inequality: 6

By identifying these decision points, I can design test cases that cover various scenarios, ensuring that the code behaves as expected under different conditions. Testing at these decision points allows me to verify the correctness of the code for both normative and exception cases, improving the overall robustness and reliability of the triangle classification code.