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

* Understandability:
  + The README.md file has a detailed overview about the requirements, setup and troubleshooting. This aids a great deal in writing unit tests and running them.
  + The inline comments present in the isTriangle.py assists in understanding the functionalities of the code and makes it intuitive to write unit tests.
* Portability:
  + The triangle project is implemented in Python, which is a portable language and requires minimal efforts in setup.
  + The code doesn’t contain any local or system dependent variables/modules which makes the project easier to write unit tests and run them without any extensive debugging.

1. 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?
   * Statement coverage percentage: 70%

A screenshot of a computer

Description automatically generated

* + Decision coverage percentage: 59%

A screenshot of a computer

Description automatically generated

* + Mutant detection rate: 23.1 %

A screenshot of a computer

Description automatically generated

1. 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, the approach to writing unit tests differed in developing a coverage-adequate tests suite and a mutation-adequate test suite.
   * In developing a coverage-adequate test suite, the focus was to write test cases in-order to ensure execution of every line/piece of code.
   * Mutation-adequate test suite development involved writing test cases to check for potential failure cases and were focused on the mutations of the code being tested.
2. Consider your mutation-adequate test suite and the triangle program. For any given program, why are some mutants not detectable?
   * The mutated code can be unreachable.
   * The mutated code involves a pre-existing source code feature making it redundant.
   * Test suit is inadequate and doesn’t handle all possible scenarios.
   * If the code involves complex changes, a mutant cannot be detectable.
3. What changes in the code coverage percentages and mutant detection rate did you observe when deleting (or commenting out) all assertions?
4. 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 can be defined as a scenario when two or more test cases involve testing the same piece of code/functionality.
* Yes, some of the test cases in our test suite can be considered redundant. The testInvalidTriangleMutant12, testInvalidTriangleMutant13 and testIsoscelesTriangleMutant1 are redundant in terms of testing the code’s functionalities.
* No, despite the test cases being redundant, we would not remove them since they are being used to test the mutated code. Despite the test cases being termed as redundant based on the definition, we would still include them to increase the testability of the code, catch mutants and reduce the risks created by code changes.