ASSIGNMENT 2

Declaration

Name	Vikrant Singh
Student no.	21262315
Program	MSc. In computing (secure software engineering)
Module Code	CA650 Software Process Quality
Assignment title	Assignment 2
Submission Date	20/03/2022
Module Coordinator	Renaat Verbruugen

I declare that this material, which I now submit for assessment, is entirely my work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offenses in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited is identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found recommended in the assignment guidelines.

Name: Vikrant Singh Date: 20/03/2022

Introduction

In this report, I'm going to be evaluating mutmut, a popular and widely used open-source mutation testing framework for python; the reason for using mutation testing is that while doing the unit test, we write test and run those tests and if they pass we believe that everything is good and we are good to go, but that's not the case so mutation testing provides better coverage for testing source code and generate better results.

Mutation Testing

Mutation testing is a technique that provides better test generation capability as we can check the source code in a better way.

So, what mutation testing does is it make some changes in the source like a change of operator and look for the exceptional behavior of the program and create exceptions called mutants for each unexpected behavior and we must work through those mutants and kill them by writing proper test cases and maybe some small changes in the source code is needed. So, we have better syntax coverage, and we can cover better the boundary value expectations with mutation testing.

So, to get full coverage we need to kill the mutants by writing appropriate tests and sometimes it's not possible to kill all the mutants and get 100 percent coverage without modifying the actual source code this will be what we have seen below while we are testing our code.

Output

It's easy to use python framework for and it has the following features:

1. It's easy to work with results as found mutants can be applied on disk with a simple command.
2. You can work incrementally because it remembers work that has been done.
3. All test runners are supported because mutmut only needs an exit code from the test command.

Installation Process

Installation of mutmut is quite simple so there are two ways to install mutmut with PyCharm.

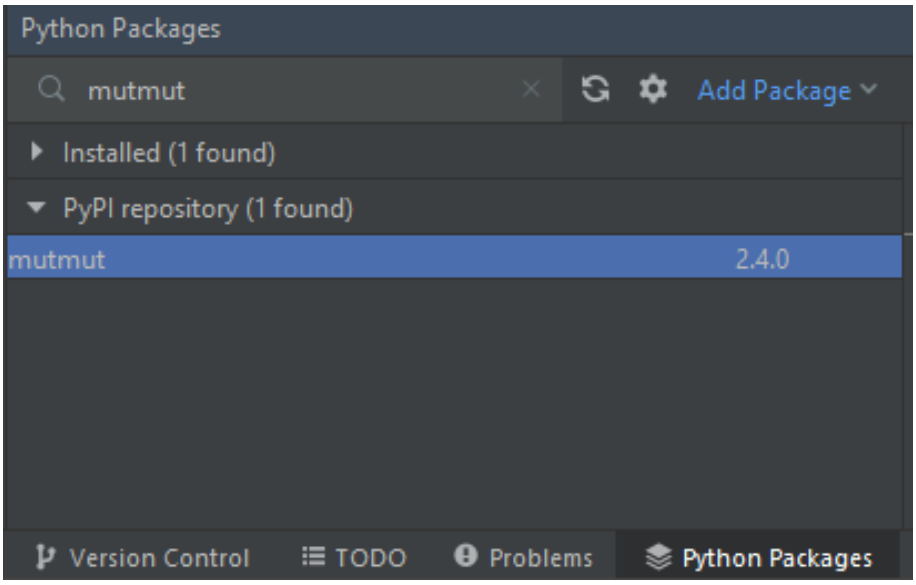
First way

Using python-pip command

```
$ pip install mutmut
```

Second way

We can also install it using package search in PyCharm itself



We can check and confirm our installation as follows:

```
PS C:\Users\vikra\PycharmProjects\mutation_testing> mutmut --help
Usage: mutmut [OPTIONS] COMMAND [ARGS]...

  Mutation testing system for Python.

Options:
  -h, --help  Show this message and exit.

Commands:
  apply       Apply a mutation on disk.
  html        Generate a HTML report of surviving mutants.
  junitxml    Show a mutation diff with junitxml format.
  result-ids  Print the IDs of the specified mutant classes (separated by...
  results     Print the results.
```

Choosing a program for testing

Now we have set up done and next task to choose a code for testing purposes, so for this purpose, I choose a very simple code to work for prime number check. The reason for choosing this extremely simple code for testing is it provides me a better understanding that how the mutation mechanism work and how we can kill mutants to get full code coverage and where its almost impossible to kill a mutant without changing the actual code.

Another reason for choosing this program is that people think about what can go wrong with such a small program and they have enough tests if something went wrong. So, to provide a better view of the capabilities of mutation testing and decencies of unit testing I choose to work with this code.

So, this is a very basic function that takes a num as a function argument and takes modulus from 2 to the number itself (which is the native prime number finding technique) to check if a number is prime or not and return false if the number is less than or equal to one is chosen which looks perfect, but we will see and write some test cases in the upcoming section to see what can go wrong with this small code and what results can be produced my unit testing and what benefit we can get using mutation testing.

```
# define a function
def isPrime(num):
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                return False
            else:
                return True

    else:
        return False
```

Testing

Let's first start by creating a test what we think is best we can create for the testing coverage

Testing case 1

```
    # test case for checking non prime nums
    def test_nonprime(self):
        self.assertEqual(isPrime(12), False)
```

As we can this test case verifies the functionality by inserting a non-prime number and False is the expected result as per the program.

Testing case 2

Now another test case we can make to test this code is by inserting a prime number

```
    # test case to check prime nums
    def test_prime(self):
        self.assertEqual(isPrime(19), True)
```

And the expected result is True in this case.

Testing case 3

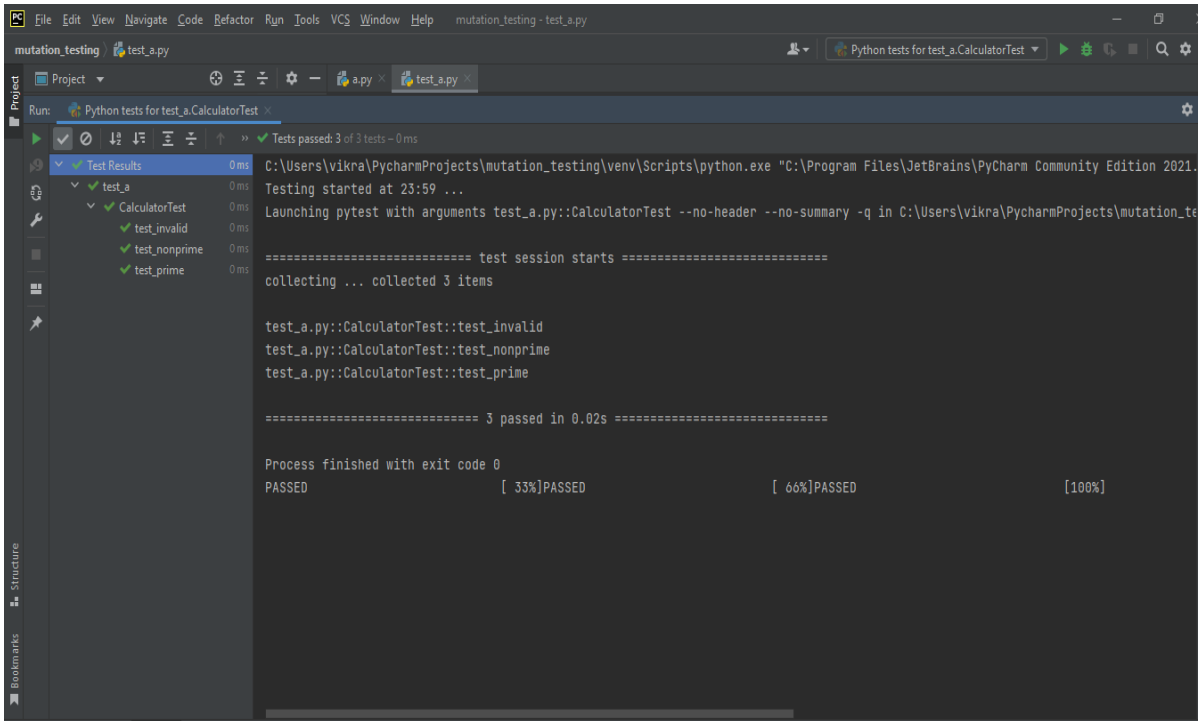
The third test we can create to check by feeding invalid like a negative number to the *isPrime ()* function.

```
    # test case to check invalid input
    def test_invalid(self):
        self.assertEqual(isPrime(-1), False)
```

So, the expected result from this test is False which is obvious as we are providing an invalid number to the function.

Results of Unit test framework

Now let's see what result our unit testing framework will produce which is pytest in this case:



As we can see by unit testing, we are getting 100 percent code coverage as all the tests get passed and now, we can think that our code is tested and good to go, before that let's just run the mutation testing against the same code and see what happens.

Results from Mutmut

Command to run mutmut for a.py

```
PS C:\Users\vikra\PycharmProjects\mutation_testing> mutmut run  --paths-to-mutate=a.py
```

```
Terminal: Windows PowerShell
Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\vikra\PycharmProjects\mutation_testing> mutmut run  --paths-to-mutate=a.py

- Mutation testing starting -

These are the steps:
1. A full test suite run will be made to make sure we
   can run the tests successfully and we know how long
   it takes (to detect infinite loops for example)
2. Mutants will be generated and checked

Results are stored in .mutmut-cache.
Print found mutants with `mutmut results`.

Legend for output:
🎉 Killed mutants.   The goal is for everything to end up in this bucket.
⏰ Timeout.          Test suite took 10 times as long as the baseline so were killed.
🙁 Survived.         This means your tests need to be expanded.
🔇 Skipped.          Skipped.

1. Using cached time for baseline tests, to run baseline again delete the cache file

2. Checking mutants
⠸ 9/9  🎉 6  ⏰ 0  🙁 0  🔇 3  🧙 0
```

As we can 9 mutants were created in such a small code and 6 of them were killed and 3 of them survived so now in the next steps let's see and try to kill these mutant to get better results.

Now let's check these mutants one by one using the following command with 1 as a mutant number.

```
PS C:\Users\vikra\PycharmProjects\mutation_testing> mutmut show 1
```

```
--- a.py
+++ a.py
@@ -1,6 +1,6 @@
 # define a function
 def isPrime(num):
-    if num > 1:
+    if num >= 1:
         for i in range(2, num):
             if (num % i) == 0:
                 return False
```

So, as we can see mutmut changes the operator of if to **>=** from **>,** and this is where the mutant is created. Now, let's write a test case to cover/kill these mutants.

The following test is added to our testing code:

```
def test_1(self):
    self.assertEqual(isPrime(1), False)
```

Similarly, look for the second mutant and kill it by writing coverage test code into the test file.

```
+++ a.py
@@ -1,6 +1,6 @@
 # define a function
 def isPrime(num):
-    if num > 1:
+    if num > 2:
         for i in range(2, num):
             if (num % i) == 0:
                 return False
```

```
def test_2(self):
    self.assertEqual(isPrime(2), True)
```

But for the third mutant its impossible to kill it without modifying our original source code as we see in the below image that it is created with loop variable value which is static and we can't access it from code so I think we are good even if we don't kill this mutant.

```
--- a.py
+++ a.py
@@ -1,7 +1,7 @@
 # define a function
 def isPrime(num):
     if num > 1:
-        for i in range(2, num):
+        for i in range(3, num):
             if (num % i) == 0:
                 return False
             else:
```

Changes in the code are needed to kill the mutant 3 so that we can get full coverage of our code but these changes are a bit useless as a variable in the range is kind of static which can't be changed by regular user input and after changes, we only left with 8 mutants instead of 9.

So, newly changed source code and whole testing code with changes are below:

New Source code: main.py

```
def isPrime(num, ran):
    if num > 1:
        for i in range(ran, num):
            if (num % i) == 0:
                return False
        else:
            return True


    else:
        return False
```

New testing code: test_main.py

```
from unittest import TestCase
from main import isPrime


# define a class
class CalculatorTest(TestCase):

    # test case for checking non prime nums
    def test_nonprime(self):
        self.assertEqual(isPrime(12, 2), False)
```

```
    # test case to check prime nums
    def test_prime(self):
        self.assertEqual(isPrime(19, 2), True)

    # test case to check invalid input
    def test_invalid(self):
        self.assertEqual(isPrime(-1, 2), False)

    # test kills mutant 1
    def test_1(self):
        self.assertEqual(isPrime(1, 2), False)

    # test kills mutant 2
    def test_2(self):
        self.assertEqual(isPrime(2, 2), True)

    # test kills mutant 3
    def test_3(self):
        self.assertEqual(isPrime(12, 3), False)
```

Report produced using mutmut

```
PS C:\Users\vikra\PycharmProjects\pythonProject1> mutmut html
```

Mutation testing report

Killed 8 out of 8 mutants

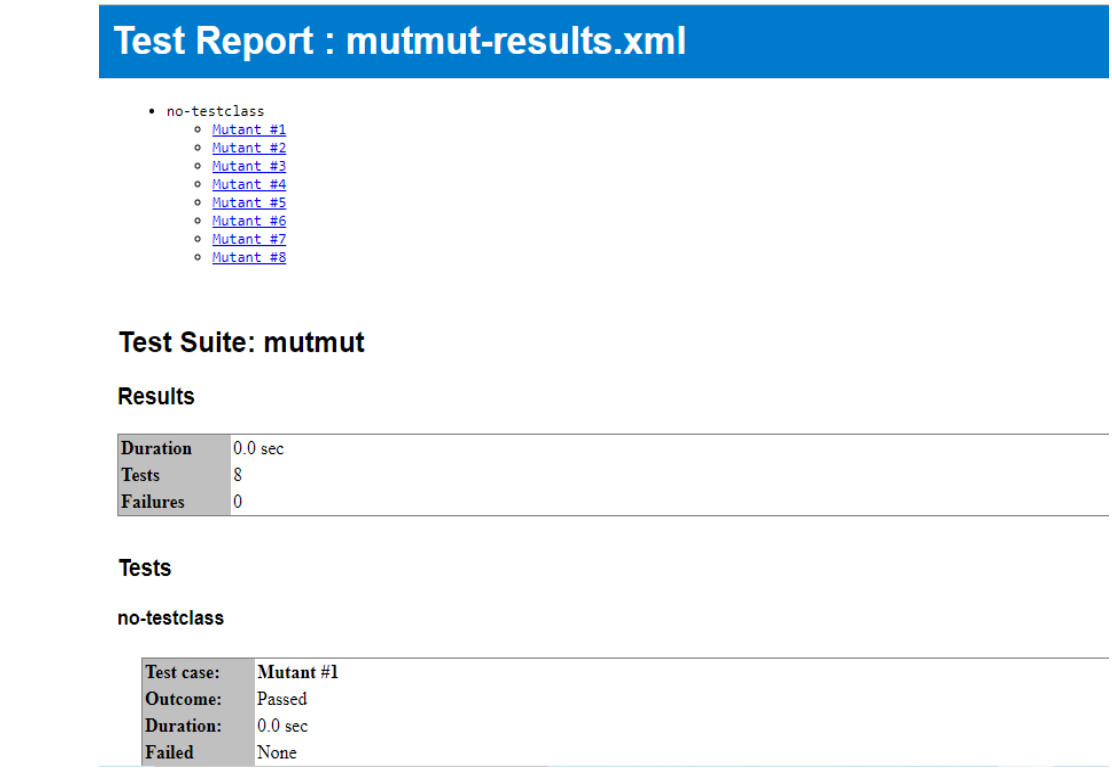
File	Total	Killed	% killed	Survived
main.py	8	8	100.00	0

This report is a bit dull so we can use the junit2html library for producing a better report And we can see every mutant detail.

```
PS C:\Users\vikra\PycharmProjects\pythonProject1> pip install junit2html
```

```
PS C:\Users\vikra\PycharmProjects\pythonProject1>  mutmut junitxml > mutmut-results.xml
```

```
PS C:\Users\vikra\PycharmProjects\pythonProject1> junit2html mutmut-results.xml mutmut-report.html
```



Test Report : mutmut-results.xml

- no-testclass
  - Mutant #1
  - Mutant #2
  - Mutant #3
  - Mutant #4
  - Mutant #5
  - Mutant #6
  - Mutant #7
  - Mutant #8

Test Suite: mutmut

Results

Duration	0.0 sec
Tests	8
Failures	0

Tests

no-testclass

Test case:	**Mutant #1**
Outcome:	Passed
Duration:	0.0 sec
Failed	None

no-testclass

Test case:	**Mutant #1**
Outcome:	Passed
Duration:	0.0 sec
Failed	None

```
None
```

Stdout

```
    if num > 1:
```

Test case:	**Mutant #2**
Outcome:	Passed
Duration:	0.0 sec
Failed	None

```
None
```

Stdout

```
    if num > 1:
```

Test case:	**Mutant #3**
Outcome:	Passed
Duration:	0.0 sec
Failed	None

Conclusion

So, overall using mutant testing is provided good syntax coverage, and also we can test boundary values quite well th3e interface of mutmut is quite handy and we just need a few commands. But, the problem with mutant testing is that I faced that it’s a very slow and time-

consuming process because it will take a very long time to scan a large code, and also it gets very expensive as we need to study and try to kill most of the mutant.

References

[1] M. Thoma, "Mutation Testing with Python," 10 Aug 2020. [Online]. Available: https://medium.com/analytics-vidhya/unit-testing-in-python-mutation-testing-7a70143180d8. [Accessed 2022].

[2] A. Hovmöller, "mutmut 2.4.0," 9 Feb 2022. [Online]. Available: https://pypi.org/project/mutmut/. [Accessed 2022].

[3] ps17, "Mutation Testing using Mutpy Module in Python," 20 Aug 2020. [Online]. Available: https://www.geeksforgeeks.org/mutation-testing-using-mutpy-module-in-python/. [Accessed 2022].