Mutation Testing on Special Numbers[3] in Various Languages

Mayank Chadha — IMT2020045
Shridhar Sharma — IMT2020065

Instructor: Prof. Meenakshi D Souza *Date: November 26, 2023*

1 Introduction to Mutation Testing

Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived.The quality of your tests can be gauged from the percentage of mutations killed.

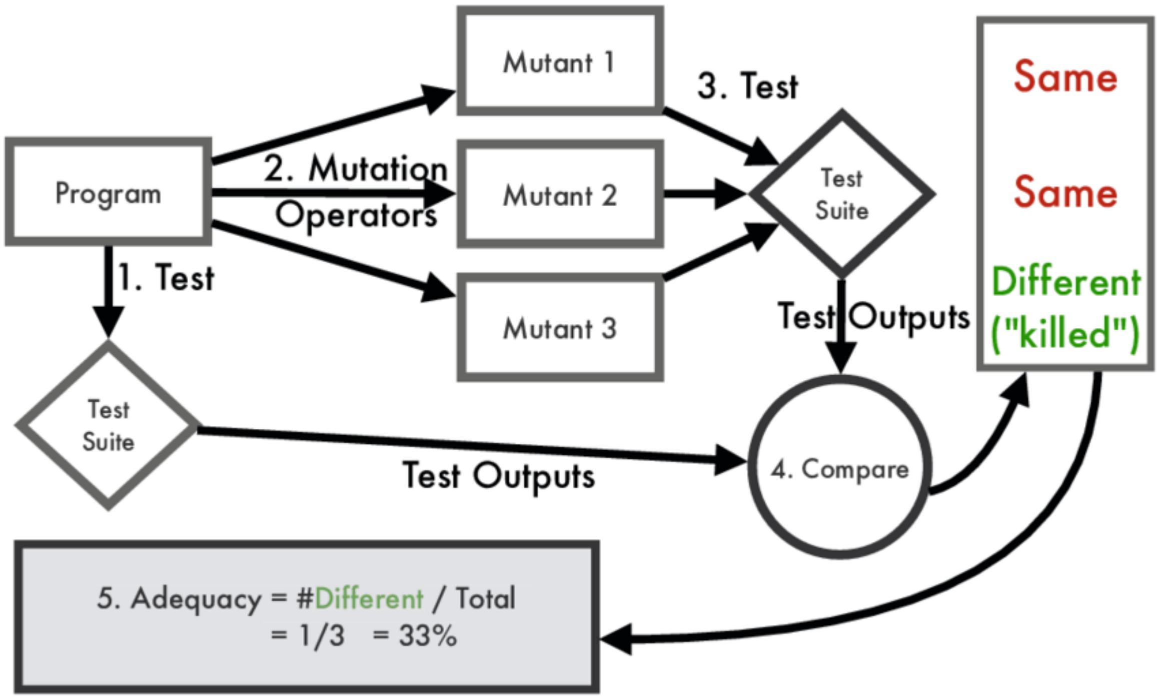


Figure 1: Overall Process of Mutation Testing

1.1 Need of Mutation Testing

Traditional test coverage (i.e line, statement, branch, etc.) measures only which code is executed by your tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested.

1.2 Tools Used

1.2.1 Python (MutPy)

Mutpy is a Mutation testing tool in Python that generates mutants and computes a mutation score. It supports the standard unittest module, generates YAML/HTML reports, and has colorful output.

1.2.2 Java (PIT)

PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the JVM. PIT runs your unit tests against automatically modified versions of your application code. When the application code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail in this situation, it may indicate an issue with the test suite. The reports produced by PIT are in an easy to read format combining line coverage and mutation coverage information.

1.3 Objective of Mutation Testing

The primary objective of mutation testing is to demonstrate the impact of these mutations by effectively “killing“ them. Killing a mutant can occur in two ways:

1. **Weak Mutant Killing:** Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to weakly kill m if the state of the execution of P on t is different from the state of execution of m immediately after l.
2. **Strong Mutant Killing:** Given a mutant $m \in M$ for a ground string program P and a test t, t is said to strongly kill m if the output of t on P is different from the output of t on m.

Our chosen testing strategy is mutation testing, specifically aiming to strongly kill mutants. This approach allows us to systematically assess the robustness of the code by ensuring that mutations induce noticeable changes in the program’s behavior, thereby enhancing the reliability of the testing process.

1.4 Mutation operators

To make syntactic changes to a program, a mutation operator serves as a guideline that substitutes portions of the source code. Given that mutations depend on these operators, scholars have created a collection of mutation operators to accommodate different programming languages, like Java. The effectiveness of these mutation operators plays a pivotal role in mutation testing.

Many mutation operators have been explored by researchers. Here are some examples of mutation operators for imperative languages:

- Statement deletion
- Statement duplication or insertion, e.g. `goto fail;`
- Replacement of boolean subexpressions with true and false
- Replacement of some arithmetic operations with others, e.g. $+$ with *, - with /
- Replacement of some boolean relations with others, e.g. $>$ with $\geq$, $==$ and $\leq$
- Replacement of variables with others from the same scope (variable types must be compatible)
- Remove method body.

1.5 Levels of Mutation Testing

- **Unit Mutation:** Unit mutation focuses on individual components like functions or methods. It delves into the internal workings of these units, making changes such as altering operators or tweaking logic. This testing method assesses how well the test suite can detect modifications within these isolated components, ensuring each piece of the software puzzle functions correctly on its own.
- **Integration Mutation:** Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components. Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered.

We will work on both of these as a part of our Mutation Testing project.

1.6 Mutation Score

The mutation score is defined as the percentage of killed mutants with the total number of mutants:

$$\text{Mutation Score} = \left(\frac{\text{Killed Mutants}}{\text{Total number of Mutants}}\right) \times 100$$

Test cases are considered mutation adequate if the score is 100%. Experimental results have shown that mutation testing is an effective approach for measuring the adequacy of test cases. However, the main drawback is the high cost associated with generating mutants and executing each test case against the mutated program.

2 Source Code

The whole Codebase was inspired by the theory written in the sixth chapter of the book Concrete Mathematics by Knuth.

Our code is a collection of functions designed to identify special types of numbers based on distinct mathematical properties. It covers a broad range of numerical categories, including prime numbers, perfect numbers, Harshad numbers, happy numbers, triangular numbers, palindromes, Armstrong numbers, perfect squares, square-free numbers, narcissistic numbers, powerful numbers, Achilles numbers, Stirling numbers, and Eulerian numbers.

The functions implement various algorithms and mathematical concepts to determine whether a given number falls into a specific category. For instance, the `isPrime` function checks if a number is prime by iterating through potential divisors. The `isHarshad` function identifies Harshad numbers by examining divisibility by the sum of their digits. The `isHappynumber` function checks for happiness by iteratively replacing numbers with the sum of squares of their digits until reaching 1 or a cycle. Each function encapsulates the unique properties that define the respective category, making the code a versatile tool for classifying and exploring different types of numbers within the realm of number theory. This collection of functions showcases the diversity and richness of number patterns, providing insights into the fascinating world of mathematical classifications.

2.1 LOC (Lines Of Code excluding documentation and inferences)

Total Lines: $\sim$ 1300 lines

1. Unit Testing in Python (MutPy)
   - `numAnalysis.py` : 285 lines
   - `utilityMethods.py` : 50 lines
   - `tester.py` : 210 lines
2. Manual Integration Testing in Python (MutPy)

- `numAnalysisIntegration.py` : 70 lines
- `utilityMethodsIntegration.py` : 20 lines
- `testerIntegration.py` : 40 lines

3. Unit and Integration Testing in Java (PIT)

   - `App.java` : 340 lines
   - `AppTest.java` : 260 lines

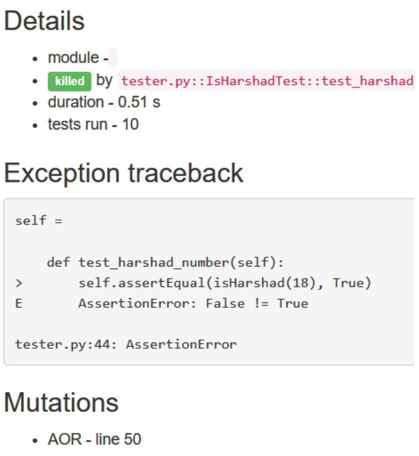
2.2 Mutation Operators Used as per Class Notes

We will only cover some of the Mutations performed since there are around 200 mutation and covering them wont be possible in this Report.

2.2.1 In Python (MutPy)

Unit Mutation Operators Generated:

- AOD - arithmetic operator deletion
- ASR - assignment operator replacement
- AOR - arithmetic operator replacement
- BCR - break continue replacement
- COI - conditional operator insertion
- LCR - logical connector replacement
- ROR - relational operator replacement
- LCR - logical connector replacement



Details

- module -
- killed by tester.py::IsHarshadTest::test_harshad
- duration - 0.51 s
- tests run - 10

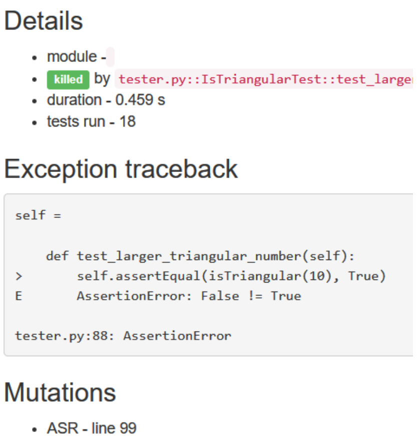
Exception traceback

```
self =

    def test_harshad_number(self):
>       self.assertEqual(isHarshad(18), True)
E       AssertionError: False != True

tester.py:44: AssertionError
```

Mutations

- AOR - line 50

Unit Mutation: Arithmetic Operator Replacement



Details

- module -
- killed by tester.py::IsTriangularTest::test_large
- duration - 0.459 s
- tests run - 18

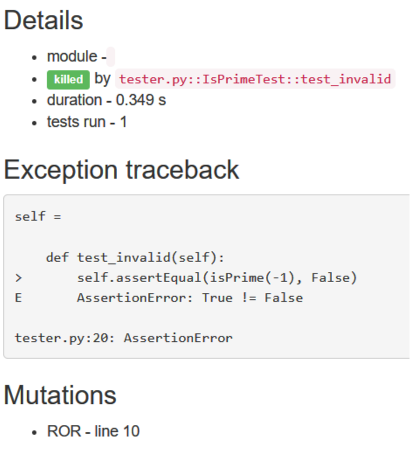
Exception traceback

```
self =

    def test_larger_triangular_number(self):
>       self.assertEqual(isTriangular(10), True)
E       AssertionError: False != True

tester.py:88: AssertionError
```

Mutations

- ASR - line 99

Unit Mutation: Assignment Operator Replacement



Details

- module -
- killed by tester.py::IsPrimeTest::test_invalid
- duration - 0.349 s
- tests run - 1

Exception traceback

```
self =

    def test_invalid(self):
>       self.assertEqual(isPrime(-1), False)
E       AssertionError: True != False

tester.py:20: AssertionError
```

Mutations

- ROR - line 10

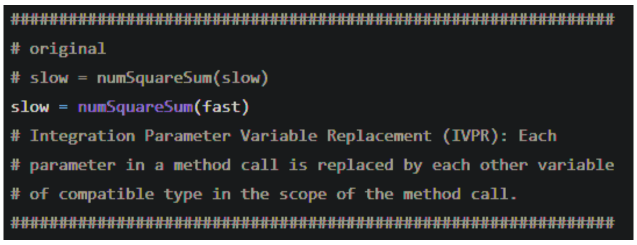
Unit Mutation: Relational Operator Replacement

Figure 2: Some Screenshots of Unit Mutation Operators in MutPy

As per the documentation, Integration Testing is not supported by MutPy so we manually tried to incorporate Mutation Testing Operators. Integrated Mutation Operators Generated:

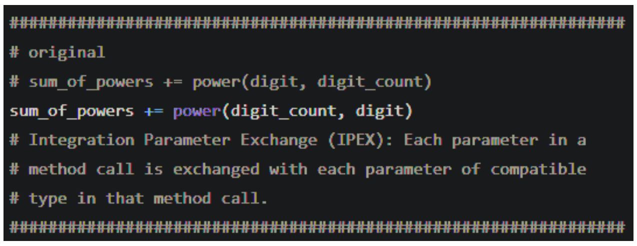
- Integration Parameter Variable Replacement (IVPR).

- Integration Parameter Exchange (IPEX).
- Integration Method Call Deletion (IMCD).
- Integration Return Expression Modification (IREM).



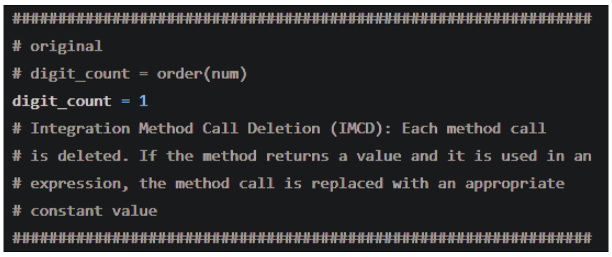
```
#################################################################
# original
# slow = numSquareSum(slow)
slow = numSquareSum(fast)
# Integration Parameter Variable Replacement (IVPR): Each
# parameter in a method call is replaced by each other variable
# of compatible type in the scope of the method call.
#################################################################
```

IVPR: `slow` is replaced by `fast`



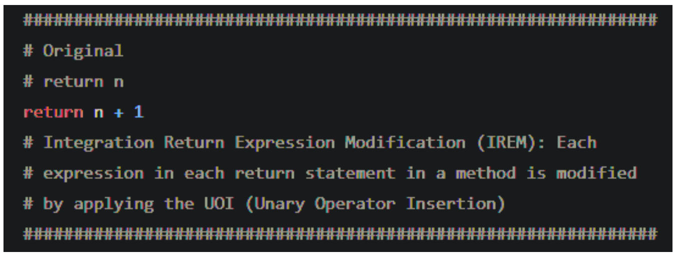
```
#################################################################
# original
# sum_of_powers += power(digit, digit_count)
sum_of_powers += power(digit_count, digit)
# Integration Parameter Exchange (IPEX): Each parameter in a
# method call is exchanged with each parameter of compatible
# type in that method call.
#################################################################
```

IPEX: `digit_count` and `digit` interchanged



```
#################################################################
# original
# digit_count = order(num)
digit_count = 1
# Integration Method Call Deletion (IMCD): Each method call
# is deleted. If the method returns a value and it is used in an
# expression, the method call is replaced with an appropriate
# constant value
#################################################################
```

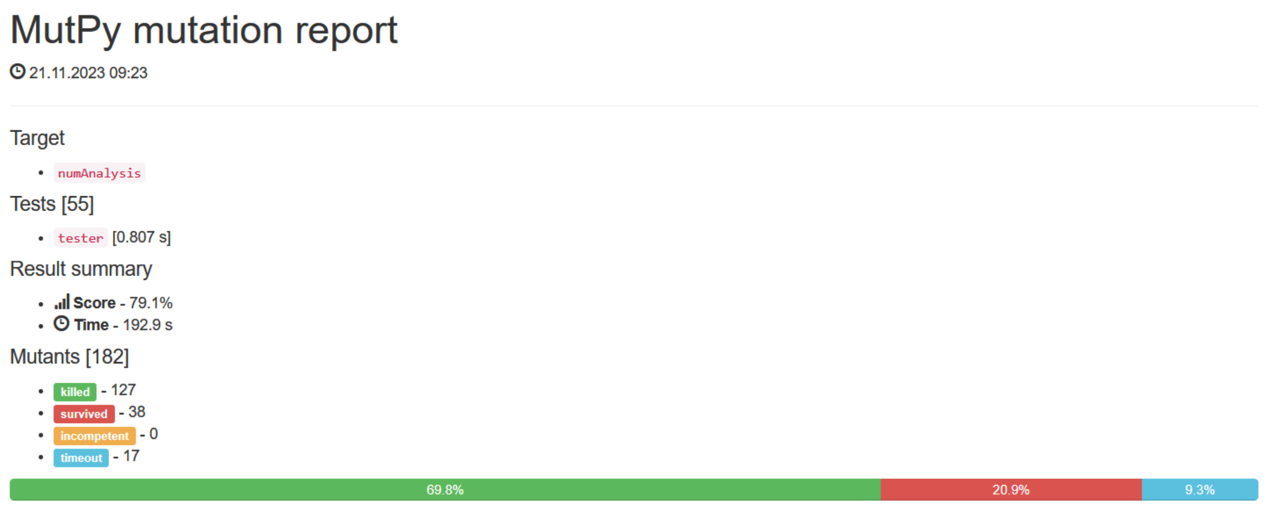
IMCD: method call deleted and replaced by 1



```
#################################################################
# Original
# return n
return n + 1
# Integration Return Expression Modification (IREM): Each
# expression in each return statement in a method is modified
# by applying the UOI (Unary Operator Insertion)
#################################################################
```

IREM: return statement modified by UOI

Figure 3: Some Screenshots of Integration Mutation Operators (Manually Added) in MutPy



MutPy mutation report

21.11.2023 09:23

Target

- numAnalysis

Tests [55]

- tester [0.807 s]

Result summary

- Score - 79.1%
- Time - 192.9 s

Mutants [182]

- killed - 127
- survived - 38
- incompetent - 0
- timeout - 17

Figure 4: Overall Unit Mutation Testing Report generated by MutPy (Python)

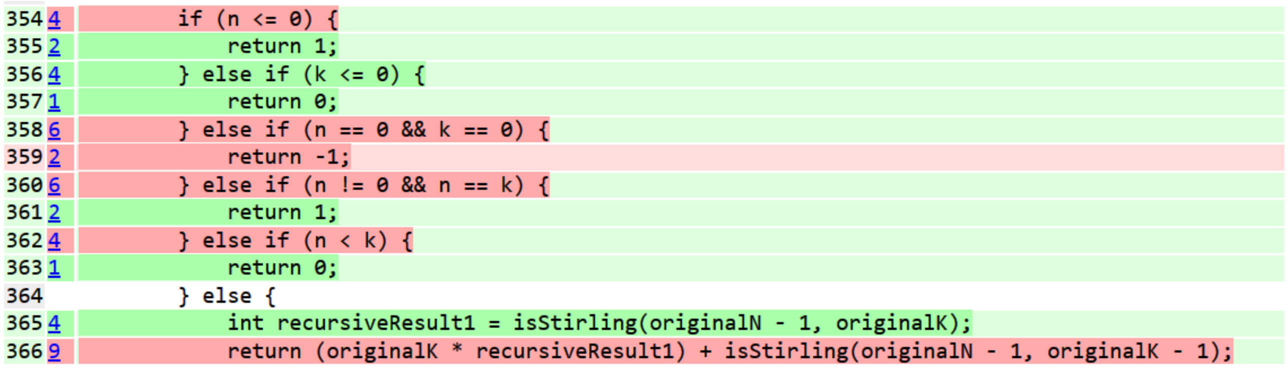
2.2.2 In Java (PIT)

Since Java Supports both Integration and Unit Mutation Testing we will discuss them together (Note in the code I have two directories one only for unit testing, the other for unit + integration testing):

Mutation Operators Generated in PIT (We are listing some of them, all of them can be seen in the figure 8):

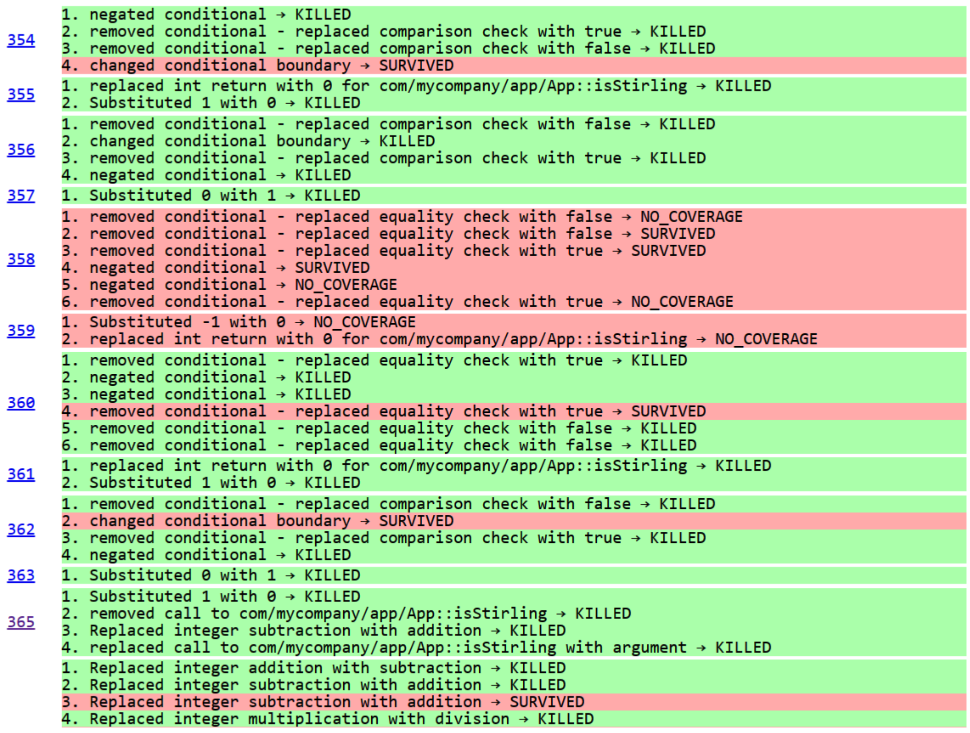
- CONDITIONALS_BOUNDARY (Relational Operator Replacement - UNIT TESTING)

- CONSTRUCTOR_CALLS (Default Constructor Deletion - INTEGRATION TESTING)
- INCREMENTS (Unary Operator Replacement - UNIT TESTING)
- MATH (Arithmetic Operator Replacement - UNIT TESTING)
- NON_VOID_METHOD_CALLS (Integration Method Call Deletion - INTEGRATION TESTING)
- PRIMITIVE_RETURNS (Integration Return Expression Modification - INTEGRATION TESTING)



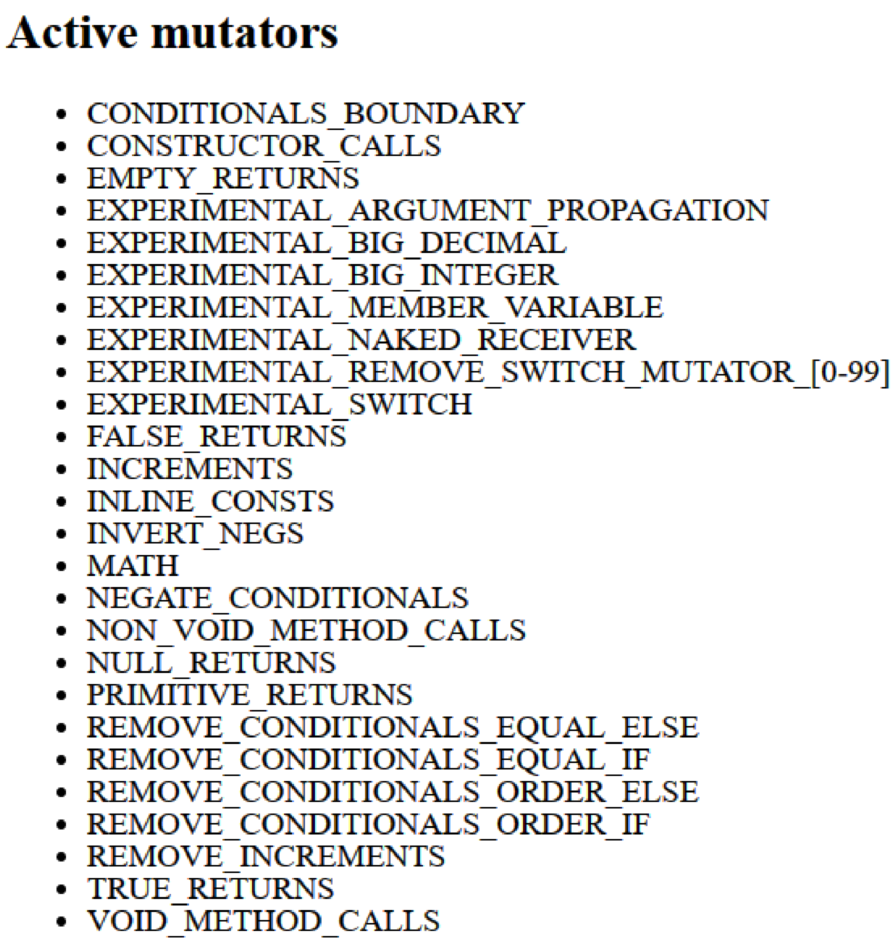
```
354 4        if (n <= 0) {
355 2            return 1;
356 4        } else if (k <= 0) {
357 1            return 0;
358 6        } else if (n == 0 && k == 0) {
359 2            return -1;
360 6        } else if (n != 0 && n == k) {
361 2            return 1;
362 4        } else if (n < k) {
363 1            return 0;
364          } else {
365 4            int recursiveResult1 = isStirling(originalN - 1, originalK);
366 9            return (originalK * recursiveResult1) + isStirling(originalN - 1, originalK - 1);
```

Figure 5: Code Structure in HTML Report in PIT (Java)



```
     1. negated conditional → KILLED
354  2. removed conditional - replaced comparison check with true → KILLED
     3. removed conditional - replaced comparison check with false → KILLED
     4. changed conditional boundary → SURVIVED
355  1. replaced int return with 0 for com/mycompany/app/App::isStirling → KILLED
     2. Substituted 1 with 0 → KILLED
     1. removed conditional - replaced comparison check with false → KILLED
356  2. changed conditional boundary → KILLED
     3. removed conditional - replaced comparison check with true → KILLED
     4. negated conditional → KILLED
357  1. Substituted 0 with 1 → KILLED
     1. removed conditional - replaced equality check with false → NO_COVERAGE
     2. removed conditional - replaced equality check with false → SURVIVED
358  3. removed conditional - replaced equality check with true → SURVIVED
     4. negated conditional → SURVIVED
     5. negated conditional → NO_COVERAGE
     6. removed conditional - replaced equality check with true → NO_COVERAGE
359  1. Substituted -1 with 0 → NO_COVERAGE
     2. replaced int return with 0 for com/mycompany/app/App::isStirling → NO_COVERAGE
     1. removed conditional - replaced equality check with true → KILLED
     2. negated conditional → KILLED
360  3. negated conditional → KILLED
     4. removed conditional - replaced equality check with true → SURVIVED
     5. removed conditional - replaced equality check with false → KILLED
     6. removed conditional - replaced equality check with false → KILLED
361  1. replaced int return with 0 for com/mycompany/app/App::isStirling → KILLED
     2. Substituted 1 with 0 → KILLED
     1. removed conditional - replaced comparison check with false → KILLED
362  2. changed conditional boundary → SURVIVED
     3. removed conditional - replaced comparison check with true → KILLED
     4. negated conditional → KILLED
363  1. Substituted 0 with 1 → KILLED
     1. Substituted 1 with 0 → KILLED
365  2. removed call to com/mycompany/app/App::isStirling → KILLED
     3. Replaced integer subtraction with addition → KILLED
     4. replaced call to com/mycompany/app/App::isStirling with argument → KILLED
     1. Replaced integer addition with subtraction → KILLED
     2. Replaced integer subtraction with addition → KILLED
     3. Replaced integer subtraction with addition → SURVIVED
     4. Replaced integer multiplication with division → KILLED
```

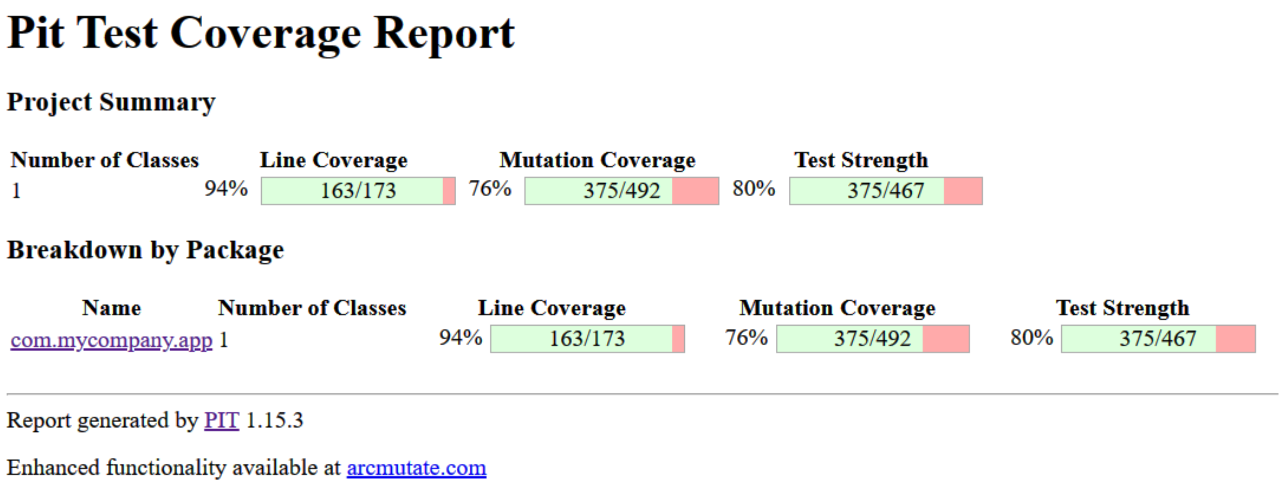
Figure 6: Mutation Operators in our Code in PIT (Java)



Active mutators

- CONDITIONALS_BOUNDARY
- CONSTRUCTOR_CALLS
- EMPTY_RETURNS
- EXPERIMENTAL_ARGUMENT_PROPAGATION
- EXPERIMENTAL_BIG_DECIMAL
- EXPERIMENTAL_BIG_INTEGER
- EXPERIMENTAL_MEMBER_VARIABLE
- EXPERIMENTAL_NAKED_RECEIVER
- EXPERIMENTAL_REMOVE_SWITCH_MUTATOR_[0-99]
- EXPERIMENTAL_SWITCH
- FALSE_RETURNS
- INCREMENTS
- INLINE_CONSTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NON_VOID_METHOD_CALLS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- REMOVE_CONDITIONALS_EQUAL_ELSE
- REMOVE_CONDITIONALS_EQUAL_IF
- REMOVE_CONDITIONALS_ORDER_ELSE
- REMOVE_CONDITIONALS_ORDER_IF
- REMOVE_INCREMENTS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Figure 7: Active Mutation Operators in our Code



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	94%	163/173	76%	375/492	80%	375/467

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.mycompany.app	1	94%	163/173	76%	375/492	80%	375/467

Report generated by PIT 1.15.3

Enhanced functionality available at arcmutate.com

Figure 8: Overall Mutation Testing Report generated by PIT (Java)

3 Comparison

In general use, Java with PIT exhibits very good documentation, typical configuration, extensive mutant operators, and supports parallel execution. On the other hand, Python with MutPy has average documentation, easy configuration, limited mutant operators, and does not support parallel execution.

Language	**Comparison On General Use**			
	Documentation	**Configuration**	**Mutants Operators**	**Parallel Execution**
Java (PIT)	Very Good	Typical	Extensive	Supports
Python (MutPy)	Average	Easy	Limited	Does Not Support

In unit testing, Java (PIT) and Python (MutPy) are compared based on runtime, types of testing available, mutants generated, and mutation score. Java (PIT) has a unit testing runtime of 104 seconds, supports both unit and integration testing, generates 204 mutants, and achieves a mutation score of 80%. Python (MutPy) has a unit testing runtime of 192 seconds, supports only unit testing, generates 182 mutants, and achieves a mutation score of 79.1%.

Language	**Comparison On Unit Testing**			
	Runtime (s)	**Types of Testing Available**	**Mutants Generated**	**Mutation Score**
Java (PIT)	104 seconds (Unit)	Unit + Integration	204	80 %
Python (MutPy)	192 seconds (Unit)	Only Unit	182	79.1 %

For integration testing, Java with PIT has a runtime of 218 seconds, supports both unit and integration testing, generates 492 mutants, and achieves a mutation score of 76%. In contrast, Python with MutPy does not provide information on integration testing runtime, supports only unit testing, and lacks data on mutants generated and mutation score.

Language	**Comparison On Integration Testing**			
	Runtime	**Types of Testing Available**	**Mutants Generated**	**Mutation Score**
Java (PIT)	218 seconds	Unit + Integration	492	76%
Python (MutPy)	-	Only Unit	-	-

To Conclude, Java's PIT excels in mutation testing with strong documentation, balanced configuration, extensive mutant operators, and support for parallel execution. It consistently achieves higher mutation scores compared to Python's MutPy, indicating superior fault detection across unit and integration testing. The tool's accessibility, coupled with its robust features, makes it a versatile choice for various project complexities.

In contrast, Python's MutPy is simpler and easier to configure, suitable for smaller projects with straightforward testing needs. However, it lacks support for parallel execution and provides no explicit information on integration testing. MutPy appears tailored for projects emphasizing unit testing. The choice between Java

(PIT) and Python (MutPy) should align with specific project requirements, considering factors like size, complexity, documentation preferences, and the need for specific testing types. This ensures the selected mutation testing tool meets the project's unique demands, fostering improved code quality and reliability.

References

[1] Mutation testing on WikiPedia

[2] Documentation of PIT (Java tool to do Mutation Testing)

[3] Source Code inspiration: Concrete Mathematics by Donald Knuth, Oren Patashnik, and Ronald Graham

[4] Types of Numbers: a glossary

[5] Documentation of MutPy (Python tool to do Mutation Testing)

[6] What is Mutation Testing? Definition from TechTarget

[7] How to Test Your Unit Tests

[8] Tables Generator