Introduction

- For this mutation testing I used MutPy, a mutation tool that is used by python. Mutation testing is used to detect whether the test cases in place can detect the changes(mutations) being made to the source code. In this case, our soure code is in the Polynomial.py and our test cases are in Polytest.py. The goal of this testing is to make sure it is able to identify the faults in the code efficiently. How MutPy works is that it creates mutated versions of the source code and then it runs the test cases against each mutated version.

List of defined mutation operators

- Arithmetic Operators - Mutate the arithmetic operators in mathematical expressions:
    - Replace '+' with '-'
    - Replace '-' with '+'
    - Replace '*' with '/'
    - Replace '/' with '*'
- Comparison Operators - Mutate the comparison operators in conditional statements:
    - Replace '==' with '!='
    - Replace '!=' with '=='
    - Replace '<' with '>'
    - Replace '>' with '<'
    - Replace '<=' with '>='
    - Replace '>=' with '<='
- Logical Operators - Mutate the logical operators in boolean expressions
    - Replace 'and' with 'or'
    - Replace 'or' with 'and'
    - Replace 'not' with an empty string(remove 'not')
- Unary Operators - Mutate unary operators like '-' and '+' in expressions
    - Replace '-' with '+'
    - Replace '+' with '-'

List of Mutant Operators That Can Be Used

- Variable Mutations - Mutate variable assignments
    - Replace a variable with another variable in the same scope
- Function Call Mutations - Mutate function calls
    - Replace function arguments with different values.

- Statement Deletion
    - Delete statements to test the robustness of the code.
- Boundary Value Mutations - Mutate boundary values.
    - Change numeric constants to slightly different values.
- Return Value Mutations - Mutate return values in functions.
    - Change the return value to a different constant or expression.
- Method Call Mutations - Mutate method calls.
    - Replace method arguments with different values.

Description of applied mutations and their impact

- Arithmetic Operator Mutation('add' method:
    - Mutation 1: Replace '+' with '-'
        - Impact: This mutation changes the addition operation to subtraction in the 'add' method. It tests whether the test suite can detect the change from addition to subtraction.
    - Mutation 2: Replace '-' with '+'
        - This mutation changes the subtraction operation to addition in the 'add' method. It tests whether the test suite can detect the change from subtraction to addition.
- Variable Mutation:
    - Mutation 3: Replace 'max_length' with another variable
        - Impact: This mutation changes the variable 'max_length' to another custom variable. It tests whether the test suite is sensitive to changes in variable names.
- Boundary Value Mutation:
    - Mutation 4: Change '0' to a slightly different value
        - Impact: This mutation changes the constant '0' to a small value('0.001 * (max_length - len(self.coefficients))'). It tests whether the test suite is robust to changes in numeric constants.
- Arithmetic Operator Mutation('sub' method):
    - Mutation 1: Replace '-' with '+'
        - Impact: This mutation changes the subtaction operation to addition in the 'sub' method. It tests whether the test suite can detect the change from subtraction to addition.
    - Mutation 2: Replace '+' with '-'
        - Impact: This mutation changes the addition operation to subtraction in the 'sub' method. It tests whether the test suite can detect the change from addition to subtraction.
- Arithmetic Operator Mutation (**mul** method):

    - Mutation 3: Replace '*' with '/'
        - Impact: This mutation changes the multiplication operation to division in the **mul** method. It tests whether the test suite can detect the change from multiplication to division.
    - Mutation 4: Replace '/' with '*'
        - Impact: This mutation changes the division operation to multiplication in the **mul** method. It tests whether the test suite can detect the change from division to multiplication.

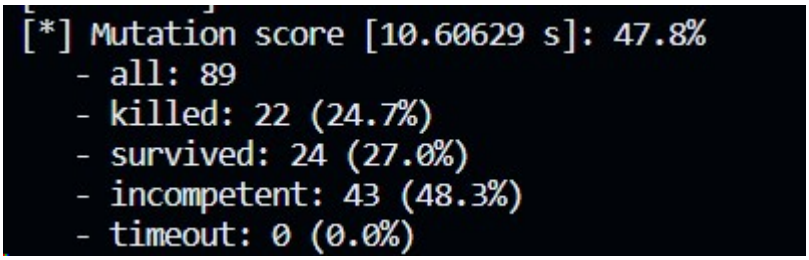
- Comparison Operator Mutation (**eq** method):
    - Mutation 5: Replace == with !=
        - Impact: This mutation changes the equality comparison to inequality in the **eq** method. It tests whether the test suite can detect the change from equality to inequality.
    - Mutation 6: Replace != with ==
        - Impact: This mutation changes the inequality comparison to equality in the **eq** method. It tests whether the test suite can detect the change from inequality to equality.

- Logical Operator Mutation (**and** method):
    - Mutation 7: Replace and with or
        - Impact: This mutation changes the logical AND operation to OR in the **and** method. It tests whether the test suite can detect the change from AND to OR.
    - Mutation 8: Replace or with and
        - Impact: This mutation changes the logical OR operation to AND in the **and** method. It tests whether the test suite can detect the change from OR to AND.

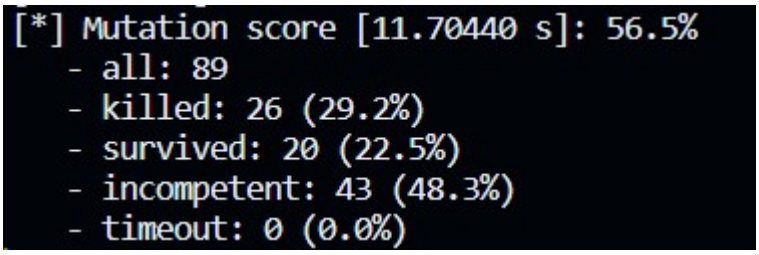
- Unary Operator Mutation (**neg** method):
    - Mutation 9: Replace - with +
        - Impact: This mutation changes the unary negation operation to unary positive in the **neg** method. It tests whether the test suite can detect the change from negation to positive.
    - Mutation 10: Replace + with -
        - Impact: This mutation changes the unary positive operation to negation in the **neg** method. It tests whether the test suite can detect the change from positive to negation.

Summary of mutant survival and killing

- Summary of Mutant Survial and Killing with original PolyTest.py



- Summary of Mutant Survival and Killing with Test Suite Revision(s):



Analysis of the test suite's effectiveness

- My mutation score went from about 47% to 56.5% which means anout 56% of my mutants were successfully identified and killed.
- Total Mutants = 89; This represents the total number of mutants generated during the testing. Each mutant represents a small change made to the source code.
- Killed Mutants = 26; The number of mutants that your test suite successfully detected and killed. These mutants represent cases where your tests identified and reported the changes introduced by the mutation.
- Survived Mutants = 20; The number of mutants that your test suite did not detect. These mutants represent cases where your tests did not identify the changes introduced by the mutation.
- Incompetent Mutants = 43; Incompetent mutants are mutants for which the testing process couldn't be completed. This might happen if the mutants cause the testing process to fail or run into an error.
- Timeout = 0; A count of 0 suggests that none of the mutants triggered a timeout during testing.

Overall Analysis: My score shows that the test suites I have does an efficient job in catching a significant amount of mutants but still has room for improvements. The number of killed mutants shows how the tests are successful in detecting and reporting changes. The number of survived mutants show that there is room for changes to improve our score. Finally, the incompetent mutants are something we got to look further into to see why the testing process did not complete. This can be caused by multiple issues such as problems with the mutation testing tool.

Recommendations for improving the test suite

- Some of the ways that we can improve the test suite are listed below:
    1. Analyze Survived Mutants - Reviewing survived mutants will help to identify the patters and commonalities among them. This can help us create more test cases to improve the coverage in areas that are currently weak.
    2. Increase Test Coverage - Find parts of the source code that has low test coverage and write test cases to cover these areas.
    3. Diverse Test Scenarios - Create test cases that cover all different ranges of inputs and conditions. This will ensure that all types of situations are handled.
    4. Refactor and Simplify Test Code - Sometimes the test code is not as readible and maintainable to make the testing easier. Refactoring and simplifying the test code so it is more organized and readible can make the process easier.
    5. Check the Mutation Score - After each new changes to the test suites or any new test suites being added, re-run the mutation testing. Check how the score has changed to see what needs to be improved or added.

Conclusion

To conclude, improving the mutation score, killed mutants, survived mutants, etc. is not an easy task. It takes a lot of trial and error and can take some time. Analyzing the test suites and results helps us to find ways to improve the numbers. Analyzing the results after each modification helps us keep track of how the results are improving or getting worse. It is important to re-run the mutation testing each time to get a clear idea of what is happening. That way we are able to add/delete what we need to get a better score. It is also important to documents your results, changes, etc. so we/others have something to go off of when the codebase is revisited to run more testing.

Some of My Added Test Cases(Suites)

```python
def test_first_degree_polynomial():
    poly = Polynomial([2, -3])  # Represents 2x - 3
    root = poly.find_root_bisection(0, 5)
    assert abs(root - 1.5) < 1e-6

def test_second_degree_polynomial():
    poly = Polynomial([1, 0, -2])  # Represents x^2 - 2
    root = poly.find_root_bisection(1, 2)
    assert abs(root - 2.0**0.5) < 1e-6

def test_third_degree_polynomial():
    poly = Polynomial([1, 0, -2, 0])  # Represents x^3 - 2x
    root = poly.find_root_bisection(-2, 2)
    assert abs(root - 0.0) < 1e-6


def test_evaluate():
    poly = Polynomial([3, 0, 2])
    assert poly.evaluate(2) == 14  # 3x^2 + 2, x=2 -> 3*4 + 2 = 14
```

```python
def test_derivative_empty_polynomial():
    poly = Polynomial([])  # Represents an empty polynomial

    # Test derivative calculation for an empty polynomial
    der = poly.get_derivative_coefficients()
    assert der == []  # Derivative of an empty polynomial should also be empty

def test_find_root_bisection_invalid_interval():
    poly = Polynomial([1, 0, -2])  # Represents x^2 - 2

    # Test bisection method with invalid interval (where root doesn't exist)
    with pytest.raises(ValueError):
        root = poly.find_root_bisection(2, 3)  # Interval does not contain a root
```

```python
def test_empty_polynomial():
    poly = Polynomial([])  # Represents an empty polynomial

    # Test string representation of an empty polynomial
    assert str(poly) == "0"

def test_negative_degree_polynomial():
    poly = Polynomial([1, 2, 3])  # Represents x^2 + 2x + 3

    # Test string representation of a polynomial with negative degree
    assert str(poly) == "1x^2 + 2x + 3"

def test_add_empty_polynomial():
    poly = Polynomial([1, 2, 3])  # Represents x^2 + 2x + 3
    empty_poly = Polynomial([])  # Represents an empty polynomial

    # Test adding an empty polynomial to another polynomial
    result = poly + empty_poly
    assert result.coefficients == [1, 2, 3]

def test_evaluate_empty_polynomial():
    poly = Polynomial([])  # Represents an empty polynomial

    # Test evaluating an empty polynomial
    assert poly.evaluate(5) == 0
```