Report

Matrix.py

The code that was used to perform testing was written from scratch and covers a few basic methods related to matrix manipulation and calculations. The code for these methods is encapsulated within a 'Matrix' class and is written in Python. The code exists in a file called matrix.py and the code shall henceforth be referenced as such in the report. The matrix.py code provides some basic matrix manipulation methods that are commonly used and provides an easy-to-use interface for method calls. The code, however, does not cater to every edge case and may or may not fail when faced with certain large or unexpected values.

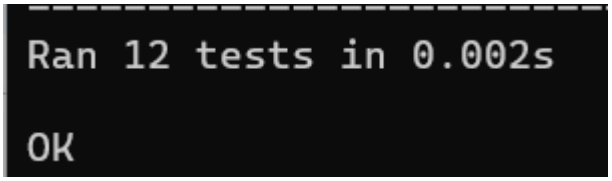
Please refer to the list below for a set of functionalities that are included in the matrix.py file:

- **Initialization:** The class constructor (__init__) initializes a matrix object with the provided matrix data. It also calculates and stores the number of rows and columns in the matrix.
- **String Representation:** The __str__ method provides a string representation of the matrix, allowing it to be printed in a readable format.
- **Addition:** The __add__ method enables addition of two matrices by element-wise addition if they have the same dimensions.
- **Subtraction:** The __sub__ method allows subtraction of two matrices by element-wise subtraction if they have the same dimensions.
- **Multiplication:** The __mul__ method enables multiplication of two matrices according to matrix multiplication rules.
- **Transpose:** The transpose method computes the transpose of the matrix.
- Determinant Calculation: The determinant method calculates the determinant of the matrix.
- **Matrix Inversion:** The inverse method computes the inverse of a square matrix.
- **Scalar Multiplication:** The scalar_multiply method multiplies the matrix by a scalar.
- **Identity Matrix:** The identity method generates an identity matrix of a given size.
- **Matrix Power:** The power method raises the matrix to a given power.
- **True Division:** The __truediv__ method divides the matrix by a scalar.
- **Matrix Equality:** The __eq__ method checks if two matrices are equal.
- **Matrix Inequality:** The __ne__ method checks if two matrices are not equal.

The following lists the set of operations that the unit tests are attempting to check and ensure that they work as intended:

- **Addition (test_addition):** Tests if matrix addition works correctly.
- **Subtraction (test_subtraction):** Tests if matrix subtraction works correctly.
- **Multiplication (test_multiplication):** Tests if matrix multiplication works correctly.
- **Transpose (test_transpose):** Tests if matrix transposition works correctly.
- **Determinant (test_determinant):** Tests if determinant calculation works correctly.
- **Inverse (test_inverse):** Tests if matrix inversion works correctly.
- **Scalar Multiplication (test_scalar_multiply):** Tests if scalar multiplication works correctly.
- **Identity Matrix (test_identity):** Tests if the identity matrix generation works correctly.
- **Matrix Power (test_power):** Tests if matrix exponentiation works correctly.
- **Division by Scalar (test_division_by_scalar):** Tests if division of a matrix by a scalar works correctly.
- **Matrix Equality (test_equality):** Tests if matrix equality comparison works correctly.
- **Matrix Inequality (test_inequality):** Tests if matrix inequality comparison works correctly.

UNITTEST RESULTS



The unitests all ran successfully and passed in each case. This gives us insights into the authenticity and reliability of the methods in matrix.py. The unittests exist in a file named ‘matrixtest.py’ and attempt to test most of the functionalities in matrix.py but do not cater to all cases.

CODE COVERAGE

File ▲	statements	missing	excluded	coverage
matrix.py	87	15	0	83%
matrixtest.py	56	0	0	100%
Total	**143**	**15**	**0**	**90%**

As can clearly be seen the code coverage testing has proved that most of the code has been tested and verified as running. The test file has also been verified as being 100% running which means that all the test cases ran and verified the results.

MUTATION TESTING

Universalmutator was also employed in the testing process so that the code can be thoroughly tested. The mutator ran on the matrixtest.py file and produced 523 valid mutants with a validity percentage of 62.4%.

```
523 VALID MUTANTS
256 INVALID MUTANTS
59 REDUNDANT MUTANTS
Valid Percentage: 62.41050119331742%
```

TSTL TESTING

TSTL testing was also performed on the code in order to have a slightly different approach from unittests so that a more reliable answer to the reliability of the matrix class can be attained. The following is an example code from the TSTL file.

```
# Matrix class TSTL test script

include <matrix.tstl>

# Test matrix addition
MATRIX = MATRIX([[1, 2], [3, 4]])
OTHER_MATRIX = MATRIX([[2, 0], [1, 2]])
result = MATRIX + OTHER_MATRIX
assert(result.matrix == [[3, 2], [4, 6]])

# Test matrix subtraction
result = MATRIX - OTHER_MATRIX
assert(result.matrix == [[-1, 2], [2, 2]])

# Test matrix multiplication
result = MATRIX * OTHER_MATRIX
assert(result.matrix == [[4, 4], [10, 8]])

# Test matrix transposition
result = MATRIX.transpose()
assert(result.matrix == [[1, 3], [2, 4]])
```

Conclusion

Finally, our examination into alternative testing methodologies for the Matrix class revealed considerable improvements in ensuring program reliability:

Unit testing with unittest allowed for thorough validation of individual components, ensuring that each method performed as expected.

Coverage testing utilizing coverage indicated the degree of code coverage, indicating areas for more testing.

Mutation testing using Universalmutator gave an alternative perspective by examining the efficacy of our test suite using mutations.

Property-based testing using TSTL allows for the verification of generic characteristics in a number of input scenarios, identifying minor defects.

Collectively, these techniques strengthen the Matrix class's resilience and reliability, highlighting the need of a comprehensive testing strategy in software development.