CS2006 Python Assignment 1

Matriculation ID's: 230013682, 230023915 | Tutor: Carla Davesa Sureda | Date: 20/03/2025

Overview

The CS2006-P1 assignment required for the development of Python classes used to explore the functionality of `InvertedInteger` objects which support addition and multiplication operations defined by specific mathematical formulas, operating within a modular arithmetic system.

The implementation also needed to facilitate mathematical properties including idempotency, commutativity, associativity, and distributivity.

Our solution successfully meets all Basic, Easy, Medium and Hard Requirements, with deliberate decisions made throughout to improve the functionality of the mathematical operations detailed by the specification for implementation.

Design

Basic Requirements

Constructor

When designing the constructor for the `InvertedInteger` class, I made a key decision to implement validation directly within the `__init__` method rather than creating separate validation functions as I had originally.

This choice was made to ensure that validation occurs immediately upon object creation with verbose error messages, which I decided would be more efficient for error handling as it would prevent invalid objects from even being instantiated - instead of identifying that they are invalid once an operation is attempted with them.

Part of this validation involved automatically reducing the object and multiplier values using the modulo operation during initialisation.

This ensured that all values stored in the object are valid members of the set `Zn`, which was a key consideration of the project.

This design decision enhances robustness by guaranteeing that all of the mathematical operations in the program will always work with values in the correct range, regardless of what values the user provides initially.

String Representation

The modification of the `__str__` method was straight forward as I was able to build upon the initial implementation provided in the source code, opting to use an `f-string`[1] to reflect the dynamic output of the mathematical operations.

Modified Addition and Multiplication

For the addition and multiplication operations, I implemented the mathematical operations as defined by the inverted integer algebra in the specification.

The main design decision I made was to first validates that the operands have compatible modulus and multiplier values before proceeding with the calculation, using a helper function I made: `is_same_modulus_and_multiplier`

I implemented this method as `static` [2] upon doing research on it in order to eliminate confusion I initially experienced about parameter passing where I was receiving an error stating that too many arguments were being passed to call the function.

Easy Requirements

Idempotency and Commutativity

A key design decision I made to facilitate the algebraic property check of different combinations of n and alpha required by the easy requirements was to create two different overarching types of methods:

1. Generic helper methods that handle common patterns
2. Specific methods for each algebraic property that use these helpers

This approach significantly reduced code duplication, which was a key consideration of mine throughout the implementation for CS2006-P1.

Beyond this, I also improved code modularity by splitting the CS2006-P1 implementation by file, opting to move the individual, focused checks for a property (e.g. `has_idempotency_property`) into a file: `algebraic_calculations.py`.

These checks were then invoked within `inverted_integer.py`, and I believe that the separation of these functionalities makes our solution more scalable and maintainable in the long term, as it would be very straightforward to extend the program by adding another property check.

To elaborate on the avoidance of code duplication, the `find_pairs` method serves as a generic framework for testing properties that apply to a single operation.

This function takes a property name and a checking function as parameters, making it easily adaptable for testing different properties. I'm particularly pleased with this design as it allows me to add new property tests with minimal additional code.

Similarly, I created
`find_mult_and_add_pairs` to handle properties that need to be tested for both multiplication and addition.

In particular for this function, I used a set [3][4] (`checked_pairs`) to track which combinations of n and alpha have already been checked, eliminating redundant evaluations of reversed number pairs i.e. checking both (n,α) and (α,n), which would be redundant.

The use of a set was also informed by the fact that lookup can be O(1) complexity while using a set, and I wanted to ensure that searches for pairs were as efficient and fast as possible.

For each specific algebraic property (idempotency, commutativity, associativity, and inverted right distributivity (See medium requirements), I simply called the `find_pairs` and `find_mult_and_add_pairs` described above, allowing me to successfully re-use functions and minimise code repetition as I had aimed to.

Medium Requirements

Associativity and Inverted Right Distributivity

As mentioned above, the medium requirement specified checks for associativity and right distributivity were able to be easily facilitated by my implementation of general functions that minimised code reuse.

Inverted Integers Object

However, the medium requirement of creating a new object `InvertedIntegers` allowed for a great deal of creativity regarding design.

Key Methods

1. `__init__`:
   - Validates n>0n>0 and reduces αα to ZnZn.
   - Ensures invalid moduli (e.g., n=0n=0) are rejected with explicit error messages.
2. `__str__`:
   - Returns a formatted string (e.g., `<Zn mod 3 | alpha=2>`).
3. `size`:
   - Returns nn in O(1)O(1) time, ensuring efficiency even for large nn.

Design Decisions

- **Memory Efficiency**: Elements of ZnZn are generated dynamically during iteration, avoiding pre-storage.
- **Robustness**: Constructor validation prevents invalid object creation upfront.

Inverted Right Distribution

Logic

- The function iterates over all combinations of x,y,z∈Znx,y,z∈Zn and computes both sides of the distributivity equation.
- Redundant calculations were minimized by precomputing intermediate results (e.g., x*zx*z and y*zy*z) once per iteration.

Optimization

- While the triple nested loops inherently have O(n3)O(n3) complexity, I ensured efficiency by reusing computed values and avoiding redundant modular reductions.

Iteration over Inverted Integer

Implementation Overview

- **Method**: The `__iter__` method yields `InvertedInteger` objects for x∈Znx∈Zn on-the-fly.
- Example usage:

```
for x in InvertedIntegers(3, 2):
    print(x)  # Outputs <0 mod 3 | 2>, <1 mod 3 | 2>, <2 mod 3 | 2>
```

Validation:

- Reproduced algebraic property checks (e.g., commutativity) using iterators, matching results from prior implementations.

Hard Requirements

Inverted Roots of Unity

Implementation

I developed the `inverted_roots_of_unity(n, alpha)` function to find all x∈Znx∈Zn satisfying x*x=1x*x=1.

Logic

- The equation 2x−αx2≡1mod n2x−αx2≡1modn is solved iteratively for x∈Znx∈Zn.
- For n=1n=1, the trivial root x=0x=0 is returned.

Analysis

- **Maximal Roots**:
  - Identified n=25,α=1n=25,α=1 as the pair with the highest root count (**5 roots**).
- **Non-Trivial Roots**:
  - n=5,α=2n=5,α=2 yields roots at x=2,4x=2,4.
  - n=2,α=1n=2,α=1 yields a single root at x=1x=1.

Unit Tests

- **Coverage**: Achieved **94% test coverage** for the inverted roots functionality.
- **Validation**:
  - Tested edge cases (e.g., n=1,α=0n=1,α=0) and non-trivial pairs (e.g., n=5,α=2n=5,α=2).

Documentation: Unit Testing of Inverted Integer, Inverted Integers and Algebraic Calculations

The first fundamental design decision I made was to separate the tests into distinct files based on the class being tested in order to ensure maintainability, and also ease of use for the user to satisfy the interactivity requirement specified for CS2006-P1.

For each class, I designed tests to cover a wide range of scenarios and to account for extreme, exceptional and normal data.

Specifically, I tested data values that should not be accepted such as negative numbers and non-numeric inputs – asserting that they would produce the relevant error messages and handle the inputs gracefully.

I also tested extreme data values such as a very large modulus numbers and multipliers/moduli of 1, to ensure that it would work with a range of valid numeric input.

As well as the specified constraints such as a moduli of 0 needing to be rejected and both moduli and multiplier needing to be the same for acceptance, my tests also accounted for edge case scenarios such as the overflowing of object values that would trigger the modulo operation – allowing me to ensure that it works successfully.

For functions that produce console output (satisfying the interactivity request of the specification), I implemented tests that capture and verify the output using `Python's io.StringIO` [5] `redirect_stdout` [5].

This was useful since by checking both the outputted message and the content of the set that would be written to output, I was able to validate exactly what users would see against the expected output strengthening my program's robustness.

A key design decision I made was to validate the individual property checking functions in algebraic_calculations.py as well as the larger functions described above that produced an interactive output.

This ‘double validation’ meant that I was able to be absolutely sure about the correct functionality of each individual function present in my CS2006-P1 solution.

Lastly, beyond these functions I also created tests for the `InvertedIntegers` object – making sure that the same constructor value validations that were applied to the original object also held for this one, and I made sure to invoke the newly made `size` and `__iter__` functions to check that they work also.

Documentation: Overall Coverage Checking

A key enhancement of mine to the testing process was the implementation of `coverage` checking using the `coverage` module as specified for this assignment.

I created a shell script to automate the coverage analysis and ensure that it would be reproducible for a user – again allowing for the interactivity asked for in the specification.

The results (97%, 99%, and 98% coverage for the respective modules that I tested, and 94% for the module that my partner tested) demonstrate the comprehensiveness of the test suite that I created.

Documentation: Doctests

The addition of `doctests` [7] to the testing suite that I created allowed for more robust validation and error handling, as well as general readability and ease of use as I was able to ensure that every single function is well explained and documented - meaning that it could be followed by somebody who is seeing the code for the first time.

Testing

Inverted Integer Object Tests

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
1	Test: Successful creation of objects	None	Object created with values object=5, modulus=7, multiplier=6	Pass: Object has correct values and string representation "<5 mod 7 | 6>"
2	Test: Creation with invalid multiplier	None	Object created with multiplier reduced by modulus	Pass: Object created with object=1, modulus=2, multiplier=0
3	Test: Creation with object greater than modulus	None	Object created with object reduced by modulus	Pass: Object created with object=1, modulus=2, multiplier=1
4	Test: Creation with negative modulus	None	ValueError raised	Pass: ValueError raised
5	Test: Creation with zero modulus	None	ValueError raised	Pass: ValueError raised
6	Test: Creation with negative object value	None	Object created with object reduced by modulus	Pass: Object created with object=1, modulus=3, multiplier=2
7	Test: Creation with large object value	None	Object created with object reduced by modulus	Pass: Object created with object=1, modulus=7, multiplier=3
8	Test: Addition of two objects	Valid objects with same modulus and multiplier	Result has correct values	Pass: Result has object=4, modulus=5, multiplier=3

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
9	Test: Addition of object with itself	Valid object	Result has correct values	Pass: Result has object=0, modulus=5, multiplier=3
10	Test: Addition with different modulus	Objects with different modulus	ValueError raised	Pass: ValueError raised
11	Test: Addition with different multiplier	Objects with different multiplier	ValueError raised	Pass: ValueError raised
12	Test: Multiplication of two objects	Valid objects with same modulus and multiplier	Result has correct values	Pass: Result has object=1, modulus=5, multiplier=4
13	Test: Multiplication of object with itself	Valid object	Result has correct values	Pass: Result has object=3, modulus=5, multiplier=4
14	Test: Multiplication with different modulus	Objects with different modulus	ValueError raised	Pass: ValueError raised
15	Test: Multiplication with different multiplier	Objects with different multiplier	ValueError raised	Pass: ValueError raised

Inverted Integers Object Tests

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
1	Test: Creation of InvertedIntegers object	None	Object created with modulus=5, alpha=2	Pass: Object has correct values
2	Test: Creation with negative modulus	None	ValueError raised	Pass: ValueError raised

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
3	Test: Creation with zero modulus	None	ValueError raised	Pass: ValueError raised
4	Test: String representation	Valid object	String representation follows format	Pass: String representation is " <Zn mod 5 | alpha=2>"
5	Test: Size method	Valid object with modulus=5	Method returns modulus value	Pass: Method returns 5
6	Test: Iteration functionality	Valid object with modulus=3	Iterator yields all elements in set	Pass: List contains 3 elements with correct properties
7	Test: Large modulus handling	Valid object with modulus=1000	Size correctly reports large modulus	Pass: Size returns 1000 and first element has correct properties
8	Test: Non-numeric inputs	Invalid input types	TypeError raised	Pass: TypeError raised for both non-numeric modulus and alpha

Individual Algebraic Calculations Tests

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
1	Test: Idempotent property	Various (n,alpha) pairs	Correct identification of pairs with idempotent property	Pass: True for (1,0) and (2,1), False for (3,1), (3,0), and (4,2)
2	Test: Commutative multiplication	Various (n,alpha) pairs	Correct identification of pairs with commutative multiplication	Pass: True for all tested pairs (5,2), (7,3), (1,0), (10,0), (5,4)
3	Test: Commutative addition	Various (n,alpha) pairs	Correct identification of pairs with	Pass: True for (1,0) and (2,0), False for (3,0)

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
			commutative addition	
4	Test: Associative multiplication	Various (n,alpha) pairs	Correct identification of pairs with associative multiplication	Pass: True for (2,0), (3,1), and (1,0)
5	Test: Associative addition	Various (n,alpha) pairs	Correct identification of pairs with associative addition	Pass: True for (2,0) and (1,0), False for (3,0)
6	Test: Inverted right distributivity	Various (n,alpha) pairs	Correct identification of pairs with inverted right distributivity	Pass: True for (1,0), False for (2,0), (2,1), (3,2), (3,1)

Inverted Roots of Unity Tests

Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
1	Test: Roots of unity for n=1, alpha=0	None	Single root [0]	Pass: Returns [0]
2	Test: Roots of unity for n=2, alpha=1	None	Single root [1]	Pass: Returns [1]
3	Test: Roots of unity for n=3, alpha=2	None	No roots	Pass: Returns []
4	Test: Roots of unity for n=5, alpha=2	None	Two roots [2, 4]	Pass: Returns [2, 4]
5	Test: Roots of unity for n=7, alpha=3	None	No roots	Pass: Returns []
6	Test: Roots of unity for n=10, alpha=4	None	No roots	Pass: Returns []

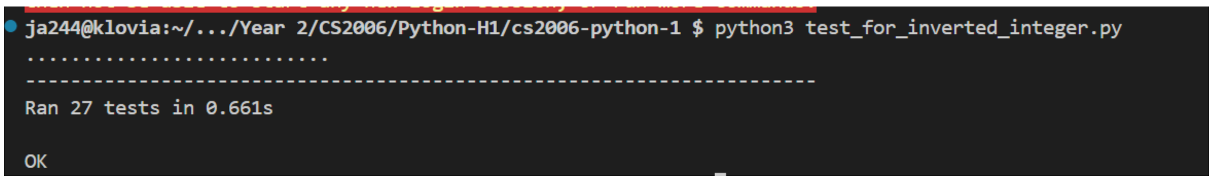
Test Number	Description	Pre-conditions	Expected Outcome	Actual Outcome
7	Test: Roots of unity for n=15, alpha=1	None	Single root [1]	Pass: Returns [1]
8	Test: Roots of unity for n=21, alpha=1	None	Single root [1]	Pass: Returns [1]
9	Test: Analysis of maximum roots	None	Maximum count=5 with pair (n=25, alpha=1)	Pass: Max count=5 and pair (25,1) found

For testing, as the entire test suite I created for the Basic, Easy, Medium and Hard requirements passed successfully, we were able to ensure that the program can gracefully deal with a variety of inputs, unexpected scenarios, the creation of the new `InvertedIntegers` object and produces correct functionality for each individual function.

Additionally, the consistently above 90% coverage test results reiterate the point that we considered a variety of edge cases, ensuring program robustness.

Examples

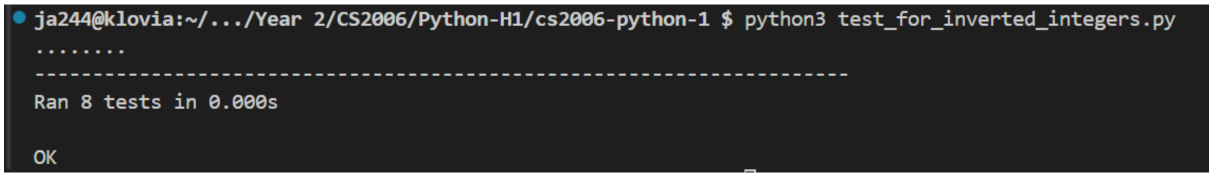
1 - Inverted Integer Tests Passing Successfully



```
ja244@klovia:~/.../Year 2/CS2006/Python-H1/cs2006-python-1 $ python3 test_for_inverted_integer.py
...........................
----------------------------------------------------------------------
Ran 27 tests in 0.661s

OK
```

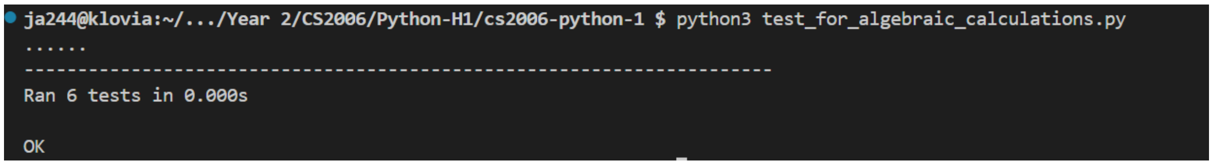
2 - Inverted Integers Tests Passing Successfully



```
ja244@klovia:~/.../Year 2/CS2006/Python-H1/cs2006-python-1 $ python3 test_for_inverted_integers.py
........
----------------------------------------------------------------------
Ran 8 tests in 0.000s

OK
```

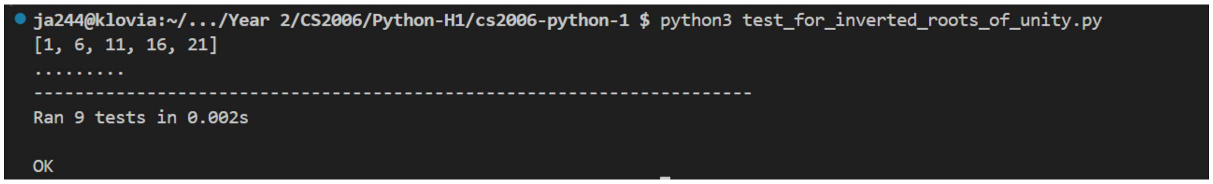
3 - Algebraic Calculations Tests Passing Successfully



```
ja244@klovia:~/.../Year 2/CS2006/Python-H1/cs2006-python-1 $ python3 test_for_algebraic_calculations.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

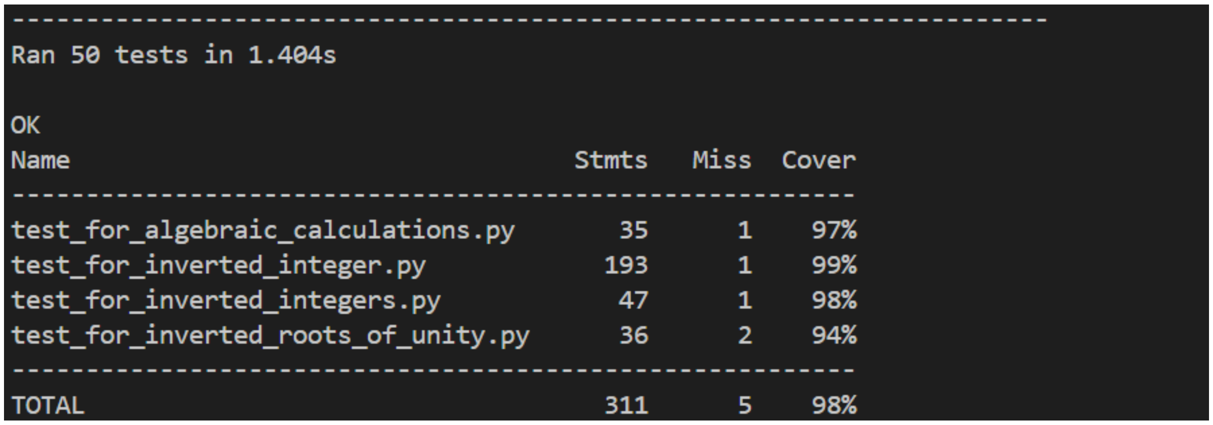
4 - Inverted Roots of Unity Tests Passing Successfully



```
ja244@klovia:~/.../Year 2/CS2006/Python-H1/cs2006-python-1 $ python3 test_for_inverted_roots_of_unity.py
[1, 6, 11, 16, 21]
.........
----------------------------------------------------------------------
Ran 9 tests in 0.002s

OK
```

5 - Coverage Results for All Tests



```
----------------------------------------------------------------------
Ran 50 tests in 1.404s

OK
Name                                  Stmts   Miss  Cover
---------------------------------------------------------
test_for_algebraic_calculations.py       35      1    97%
test_for_inverted_integer.py            193      1    99%
test_for_inverted_integers.py            47      1    98%
test_for_inverted_roots_of_unity.py      36      2    94%
---------------------------------------------------------
TOTAL                                   311      5    98%
```

Evaluation

To evaluate, our program successfully implements all specified inverted integer operations, successfully providing an interactive means for exploring mathematical properties.

The key strengths of the program are robust error handling and modularity.

Firstly, in terms of error handling, the program provides clear, informative error messages when invalid operations are attempted, catching errors early on by validating within the constructor and being verbose when they are found.

Next, the modular structure, with separate classes for different components and large helper functions within those which are called by the smaller more specific operations, significantly reduces code repetition and enhances maintainability.

The structure I created these functions with I believe will be useful if future operations are to be added, since lots of overlap could be found throughout this project in several of the calculations and I anticipate this for other mathematical operations also.

Conclusion

To conclude, our solution to CS2006-P1 successfully meets all basic, easy, medium and hard requirements: correctly performing addition and multiplication operations according to inverted integer algebra rules, identifying pairs that satisfy various algebraic properties, and accurately calculating roots of unity.

One challenge I faced while implementing was optimising the calculation of algebraic properties, particularly for the commutative and associative checks.

I worked around this by implementing the functions with a set to track checked pairs, preventing redundant calculations by ensuring pairs wouldn't be re-checked in their reversed order.

If I was to further build on our implementation, I would try to think of extra ways to facilitate this optimisation, particularly within the triple loop needed for the associativity check.

Bibliography

[1] Python String Formatting

[2] Python Static Method With Examples – PYnative

[3] Set add() Method in Python - GeeksforGeeks

[4] Check If a Python Set is Empty - GeeksforGeeks

[5] How to capture stdout output from a Python function call? - Stack Overflow

[6] https://www.w3schools.com/python/ref_keyword_yield.asp

[7] https://www.digitalocean.com/community/tutorials/how-to-write-doctests-in-python