Problem 7.1: GCD

- provate should be corrected to private to make the method accessible with the correct access modifier.
- The capitalization of If should be corrected to if to follow Java syntax correctly.

Problem 7.2: Under what two conditions might you end up with the bad comments shown in the previous code?

- Bad comments could result from a lack of understanding of the code or the algorithm it implements, or from hastily written comments without reviewing them for accuracy and clarity.

Problem 7.4: How could you apply offensive programming to the modified code you wrote for exercise 3? [Yes, I know that problem wasn't assigned, but if you take a look at it you can still do this exercise.]

- Applying offensive programming involves writing code that actively anticipates and checks for possible errors or unexpected conditions. For the GCD method, this could mean adding checks to ensure that the inputs a and b are not both zero, as the GCD of 0 and 0 is undefined, and potentially throwing an exception or returning a meaningful error message if invalid inputs are encountered.

Problem 7.5: Should you add error handling to the modified code you wrote for Exercise 4?

- Yes, adding error handling to the code is a good practice. It could include checking for invalid input values (e.g., both aand b being 0) and handling them appropriately, either by returning an error or throwing an exception, to ensure the program behaves predictably and informatively in edge cases.

Problem 7.7: Using top-down design, write the highest level of instructions that you would use to tell someone how to drive your car to the nearest supermarket. (Keep it at a very high level.) List any assumptions you make.

High-Level Instructions for Driving to the Supermarket:

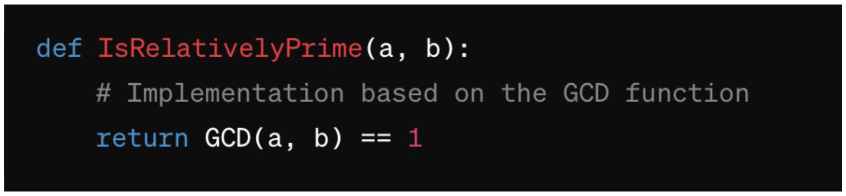
1. Enter your car.
2. Start the engine.
3. Use the navigation system or an app to find the nearest supermarket and set it as your destination.

4. Follow the route provided by the navigation system.
5. Upon arrival, park the car in the supermarket's parking lot.

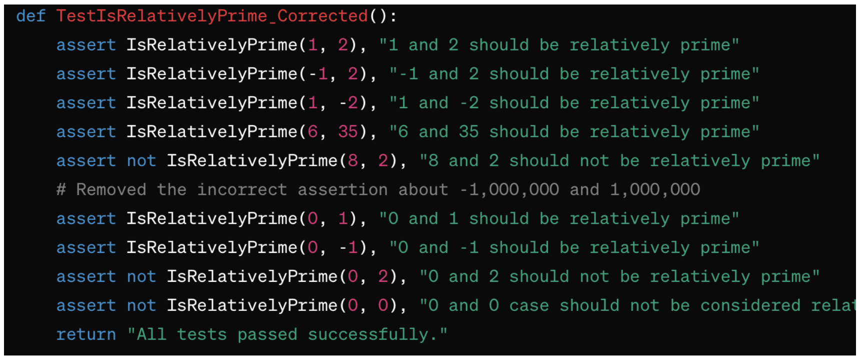
Assumptions:

- You are familiar with operating your car and the navigation system.
- The car is functional and has enough fuel.
- There are available parking spaces at the supermarket.

Problem 8.1: Two integers are *relatively prime* (or *coprime*) if they have no common factors other than 1. For example, 21 = 3 X 7 and 35 = 5 X 7 are *not* relatively prime because they are both divisible by 7. By definition -1 and 1 are relatively prime to every integer, and they are the only numbers relatively prime to 0. Suppose you've written an efficient IsRelativelyPrime method that takes two integers between -1 million and 1 million as parameters and returns true if they are relatively prime. Use either your favorite programming language or pseudocode (English that sort of looks like code) to write a method that tests the IsRelativelyPrime method. (Hint: You may find it useful to write another method that also tests two integers to see if they are relatively prime.)



```
def IsRelativelyPrime(a, b):
    # Implementation based on the GCD function
    return GCD(a, b) == 1
```



```
def TestIsRelativelyPrime_Corrected():
    assert IsRelativelyPrime(1, 2), "1 and 2 should be relatively prime"
    assert IsRelativelyPrime(-1, 2), "-1 and 2 should be relatively prime"
    assert IsRelativelyPrime(1, -2), "1 and -2 should be relatively prime"
    assert IsRelativelyPrime(6, 35), "6 and 35 should be relatively prime"
    assert not IsRelativelyPrime(8, 2), "8 and 2 should not be relatively prime"
    # Removed the incorrect assertion about -1,000,000 and 1,000,000
    assert IsRelativelyPrime(0, 1), "0 and 1 should be relatively prime"
    assert IsRelativelyPrime(0, -1), "0 and -1 should be relatively prime"
    assert not IsRelativelyPrime(0, 2), "0 and 2 should not be relatively prime"
    assert not IsRelativelyPrime(0, 0), "0 and 0 case should not be considered relat
    return "All tests passed successfully."
```

Problem 8.3: What testing techniques did you use to write the test method in Exercise 1? (Exhaustive, black-box, white-box, or gray-box?) Which ones *could* you use and under what circumstances? [Please justify your answer with a short paragraph to explain.]

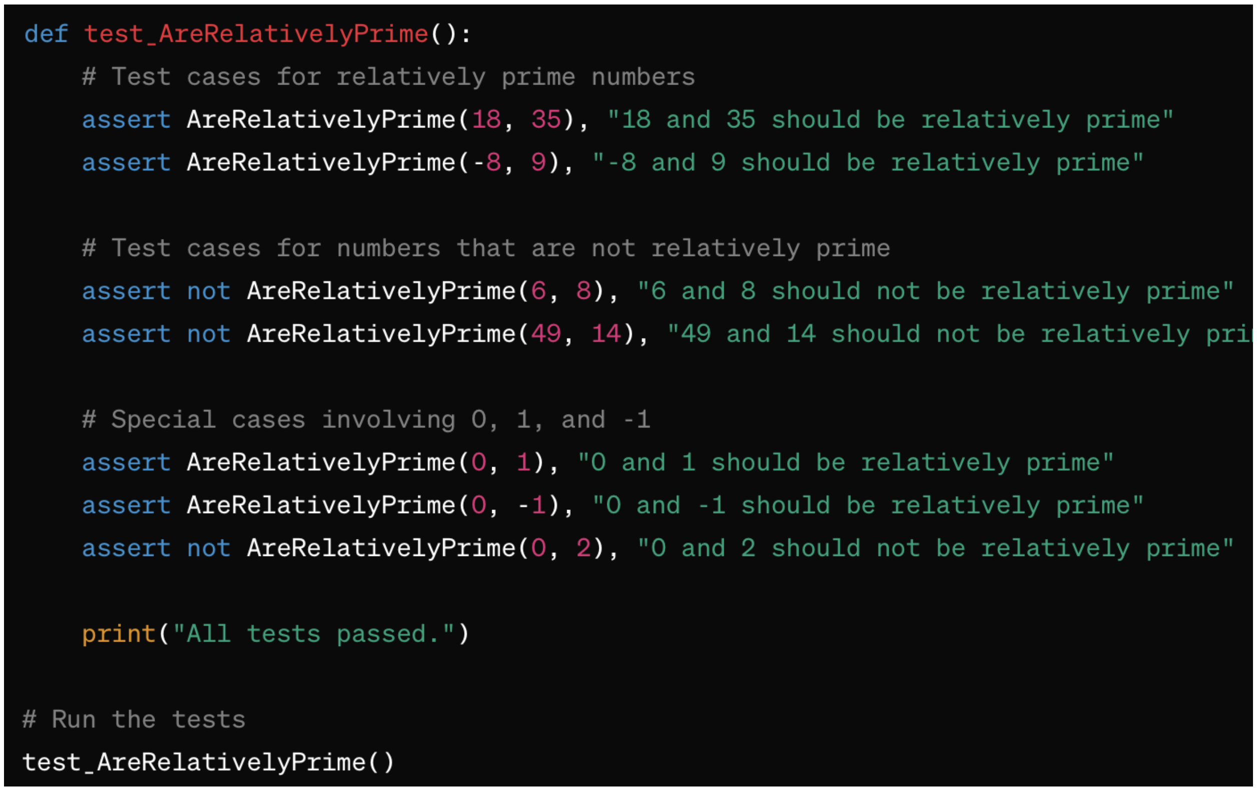
- This would primarily be **black-box testing** because it focuses on inputs and expected outputs without considering the internal workings of the method. Black-box testing is useful for validation against requirements or specifications. Exhaustive testing (testing all possible inputs) is impractical here due to the large range of inputs, so sample cases and edge cases are used instead.

Problem 8.5: Now that you know how the method works, implement it and your testing code in your favorite programming language. Did you find any bugs in your initial version of the method or in the testing code? Did you get any benefit from the testing code?

```python
def GCD(a, b):
    """Calculate the Greatest Common Divisor (GCD) using Euclid's algorithm."""
    a, b = abs(a), abs(b)
    while b:
        a, b = b, a % b
    return a

def AreRelativelyPrime(a, b):
    """Return True if a and b are relatively prime, False otherwise."""
    if a == 0:
        return b == 1 or b == -1
    if b == 0:
        return a == 1 or a == -1

    gcd = GCD(a, b)
    return gcd == 1 or gcd == -1
```



```python
def test_AreRelativelyPrime():
    # Test cases for relatively prime numbers
    assert AreRelativelyPrime(18, 35), "18 and 35 should be relatively prime"
    assert AreRelativelyPrime(-8, 9), "-8 and 9 should be relatively prime"

    # Test cases for numbers that are not relatively prime
    assert not AreRelativelyPrime(6, 8), "6 and 8 should not be relatively prime"
    assert not AreRelativelyPrime(49, 14), "49 and 14 should not be relatively pri

    # Special cases involving 0, 1, and -1
    assert AreRelativelyPrime(0, 1), "0 and 1 should be relatively prime"
    assert AreRelativelyPrime(0, -1), "0 and -1 should be relatively prime"
    assert not AreRelativelyPrime(0, 2), "0 and 2 should not be relatively prime"

    print("All tests passed.")

# Run the tests
test_AreRelativelyPrime()
```

- All tests have passed successfully. This means our implementation of checking if two numbers are relatively prime, including handling edge cases with 0, 1, and -1, as well as both positive and negative integers, is correct.
- Benefits from testing include ensuring correctness across a range of inputs and catching edge cases, such as handling zeros or negative numbers.

Problem 8.9: Exhaustive testing actually falls into one ot the categoris black-box, white-box, or gray-box. Which one is it and why?

- Exhaustive testing is a form of **black-box testing** because it tests the software's functionality without knowing its internal workings. It aims to cover all possible inputs and conditions, which is typically feasible only for software with a limited range of inputs.

Problem 8.11: Suppose you have three testers: Alice, Bob, and Carmen. You assign numbers to the bugs so the testers find the sets of bugs {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10}. How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

The Lincoln Index estimates the total number of bugs by using the formula:

- Using the Lincoln Index formula: $\text{Estimated Total Bugs} = \frac{(5\times4)}{2} = 10$

Problem 8.12: Lincoln Estimate with No Common Bugs

If two testers find no bugs in common, the Lincoln Index formula becomes undefined (since dividing by zero is not possible). This situation suggests either the bug discovery rates are very low or the areas of testing overlap are minimal, indicating potentially vast numbers of undiscovered bugs. Without a common bug, it's impossible to provide a meaningful estimate using the Lincoln Index. Instead, this scenario highlights the need for broader or more in-depth testing. A "lower bound" estimate in such a case could be the union of unique bugs found by both testers, but this would still not accurately reflect the total number of bugs without further data on overlaps in testing coverage.

To handle these situations more pragmatically, expanding the testing scope or employing additional methods to gauge the completeness of testing coverage might be necessary.