Final Triangle.py HW02a

The objective of this assignment is to (a) develop a set of tests for an existing triangle classification program, (b) use those tests to find and fix defects in that program, and (c) report on your testing results for the Triangle problem. You will start with an existing implementation of the classify triangle program that will be given to you and a starter test program that tests the classify triangle program, but those tests are not complete. In order to determine if the program is correctly implemented, you will need to update the set of test cases in the test program. You will need to update the test program until you feel that your tests adequately test all of the conditions. Then you should run the complete set of tests against the original triangle program to see how correct the triangle program is. Report on those results in a formal test report.

Sarah Wiessler

Summary

The code now passes all 11 test cases. My strategy for determining an adequate number of tests included making sure the triangles would be identified correctly with correct input and checking that invalid input would be caught. I attempted to eliminate redundancy, so that the issue would be clear when test cases failed. I was able to find and fix an error in my initial test cases, where the integer constraint was violated.

The process of documenting the incremental defect fixes through test runs was very helpful to identify personal debugging techniques, and I enjoyed documenting the defects I found. Learning about automatic testing is absolutely essential and I am glad I had the opportunity to do it through this assignment. Additionally, I believe the concept of fixing the code instead of erasing everything and inserting your own logic is incredibly relevant, and will absolutely be a part of my career.

I pledge my honor that I have abided by the Stevens Honor System.

Detailed Results

Techniques

I diagnosed defects in the code for the first three test runs by implementing a main method in Triangle.py and setting breakpoints at every “if” statement, then using the Visual Studio Code debugger to step through the code to determine exactly where the code went wrong. For the last two test runs I used the full test file to diagnose the issues. I started by testing one case in order to diagnose the input checking issues. I moved on to checking if the program would correctly identify each type of triangle. Finally, I checked it against the full panel of tests.

Constraints

Constraints applied by the program stated that input must be and integers need to be within 0-200 range. Additionally, the sum of any two sides must be less than the third.

Data Inputs

Integers, strings, list, float

Results

Table 1: Results

Test ID	Input	Expected Results	Actual Result	Pass or Fail
testInvalidInputA	“zoinks”,4,5	NotATriangle	NotATriangle	Pass
testInvalidInputD	[20],4,4	NotATriangle	NotATriangle	Pass
testEquilateralTriangleA	4,4,4	Equilateral	Equilateral	Pass
testInvalidInputB	-1,4,4	NotATriangle	NotATriangle	Pass
testInvalidInputC	0,4,4	NotATriangle	NotATriangle	Pass
testInvalidInputE	10.5,10.5,20	NotATriangle	NotATriangle	Pass
testIsocelesTriangleA	11,11,20	Isoceles	Isoceles	Pass
testNonviableSideA	3,4,8	NotATriangle	NotATriangle	Pass
testRightTriangleA	3,4,5	Right	Right	Pass
testRightTriangleB	5,3,4	Right	Right	Pass
testScaleneTriangleA	10,15,20	Scalene	Scalene	Pass

Table 2: Test Runs

	Test Run 1	Test Run 2	Test Run 3	Test Run 4	Test Run 5
Tests Planned	**1**	**1**	**4**	**11**	**11**
Tests Executed	**1**	**1**	**4**	**11**	**11**
Tests Passed	**0**	**0**	**3**	**9**	**11**
Defects Found	**3**	**2**	**1**	**2**	**0**
Defects Fixed	**2**	**1**	**3**	**2**	**0**

Table 3: Defect vs. Modification

Defect	Modification
if a <= 0 or **b <= b** or c <= 0: return 'InvalidInput'	if a <= 0 or **b <= 0** or c <= 0: return 'InvalidInput'
InvalidInput	NotATriangle

if (a >= (b - c)) or (b >= (a - c)) or (c >= (a + b)): return 'NotATriangle'	if (a >= (b + c)) or (b >= (a + c)) or (c >= (a + b)): return 'NotATriangle'
elif ((a * 2) + (b * 2)) == (c * 2): return 'Right'	elif ((a ** 2) + (b ** 2)) == (c ** 2): return 'Right'
if a == b and b == a: return ‘Equilateral’	if a == b and b == c: return ‘Equilateral’
	Moved isInstance checks above rest of input checks to weed out all other data types
elif ((a ** 2) + (b ** 2)) == (c ** 2): return 'Right'	elif (((a ** 2) + (b ** 2)) == (c ** 2)) or (((b ** 2) + (c ** 2)) == (a ** 2)) or (((a ** 2) + (c ** 2)) == (b ** 2)): return 'Right'
NotATriangle!=Isoceles test case, failed integer constraint	Changed 10.5 isosceles test case to NotATriangle