|  |  |
| --- | --- |
| **CM2010 Software design and development** | |
|  | |
| **Student Name:** | Owen Lee Wei Hern |
| **Date Submitted:** | 16th December 2022 |
| **Degree Title:** | Computer Science |
| **Local Institution:** | Singapore Institute of Management |
| **Student ID:** | 220218799 |

**Part 2**

For this part of the coursework, I have chosen snakestats as my unit test example program.

**Test Set 1: Target function name**

The function I will be testing for set 1 is my **getMin()** function. This functions searches through a list of numbers and returns the smallest number out of all the numbers in the list.

**Test Set 1: Explanation of strategy**

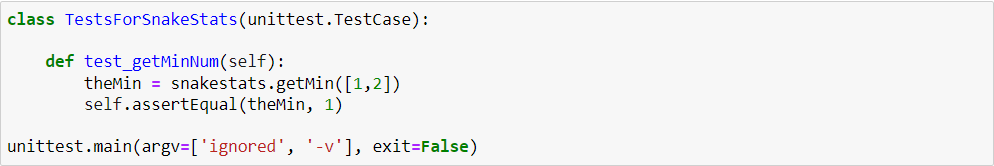
My strategy for this test set is to check against a bare minimum of 2 values and see if the minimum is returned. Secondly, I will expand the set of numbers to a range of N and see if the minimum can be returned. Finally, I will be testing against strings as well and make sure the minimum can still be returned.

**Test Set 1: Test 1**

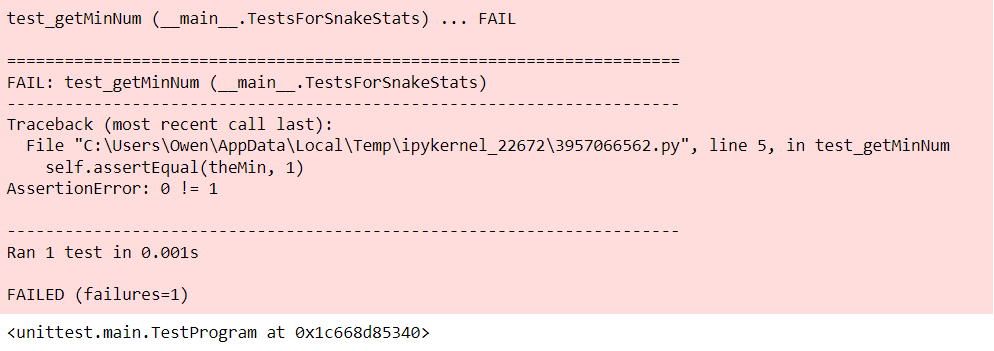
This is the current code for the **getMin()** function.



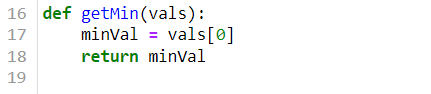
In this test, I am testing to see if the snakestats function **getMin()** will return me the smallest value. The values that I am comparing are 1 and 2. In this case, the number 1 should be returned.

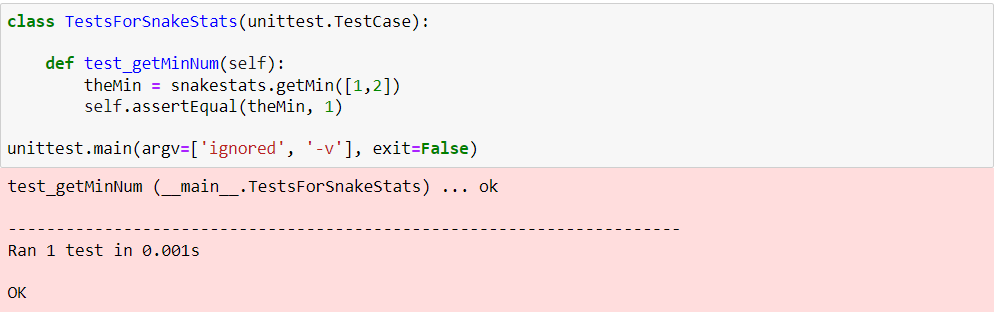


Running this test against the **getMin()** function, we run into an assertion error, **0 != 1**.



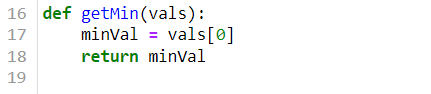
By changing the **getMin()** function, it should now return 1 instead, which is the minimum value.



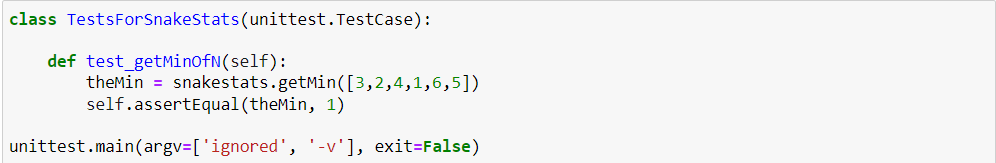
Running the same test again, we see that it indeed passes the test. 

**Test Set 1: Test 2**

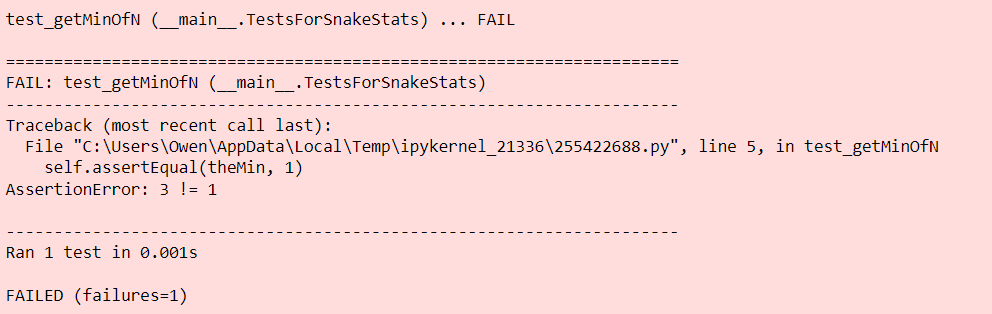
The current code for **getMin()**.



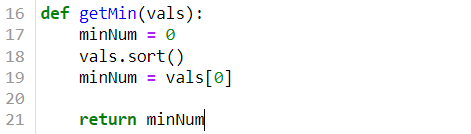
In this test, I am testing if the function is able to return me the smallest value of N list of numbers.



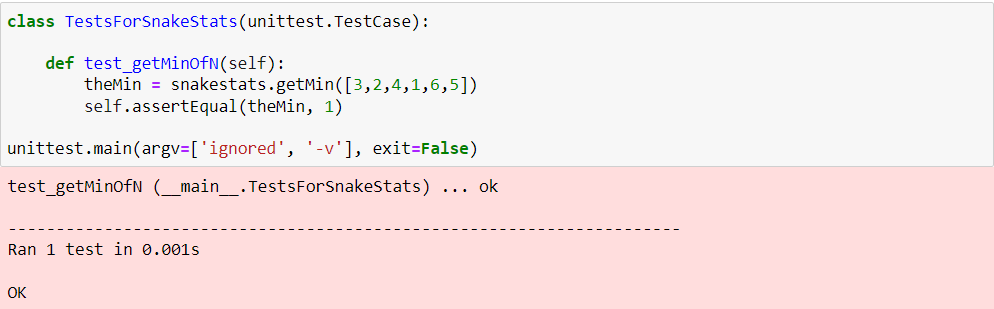
Running the test against the current code in **getMin()**, we see that we encounter another assertion error, **3 != 1**.



By changing the **getMin()** function, it will now sort the list in ascending order and return the value in index 0.

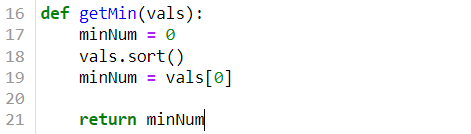


If the same test is run again, we see that it is now passing the test.

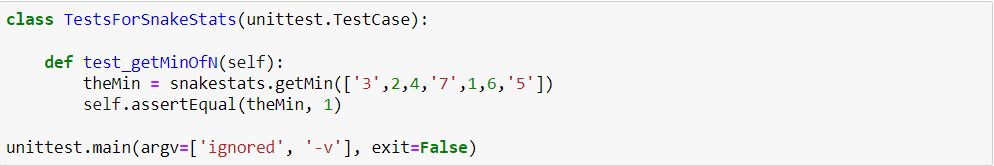


**Test Set 1: Test 3**

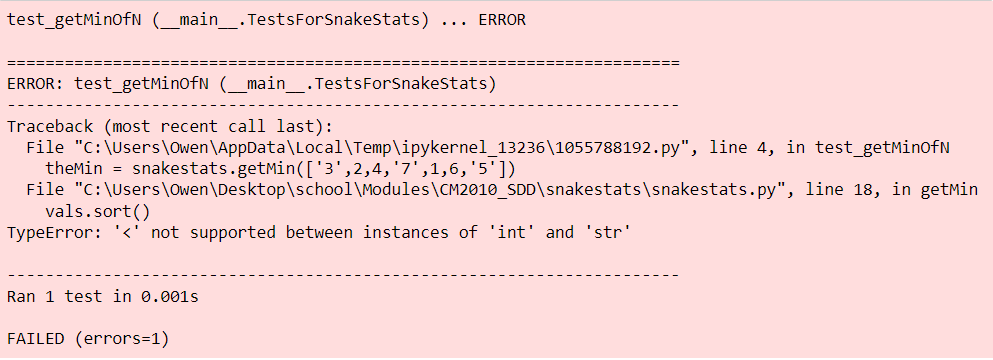
The current code for **getMin()**.



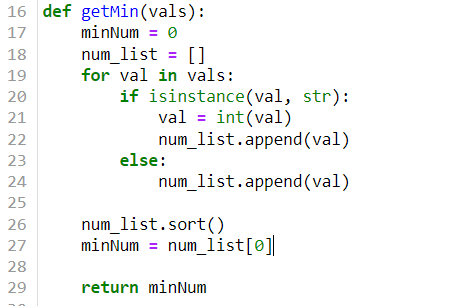
In this test, **getMin()** should be able to convert strings of integers into integers for sorting and fetch the minimum value.



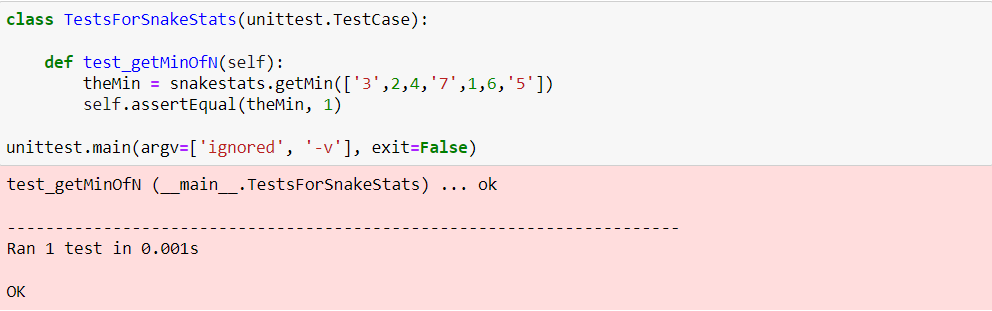
Running this test against the function, we see that we encounter a type error, that the sort() function is incompatible between strings and integers.



By changing the **getMin()** to do a type conversion beforehand and push it into a new list for sorting, it should now be able to return as the min value.



Running the test against the newest iteration of **getMin()**, we see that the test is now indeed passing.



**Test Set 2: Target function name**

The function I will be testing for Set 2 will be my **getFreq()** function. This function should loop through a list of integers, storing the count of the number of times they appear in the list. When it is done looping, the integer that appeared most frequently/the mode should be returned, if the frequency for all are the same, **True** should be returned instead.

**Test Set 2: Explanation of strategy**

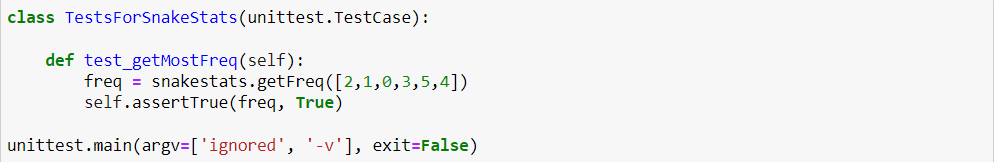
My strategy for testing is to first check if the function works when checking against a list of numbers with the same frequency, a list of numbers with different frequencies and how can I expand this test to check against a list of doubles as well.

**Test Set 2: Test 1**

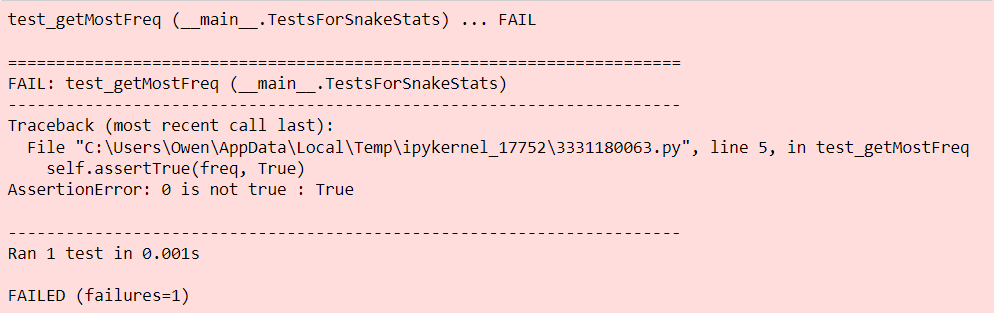
This is the current code for my **getFreq()** function.



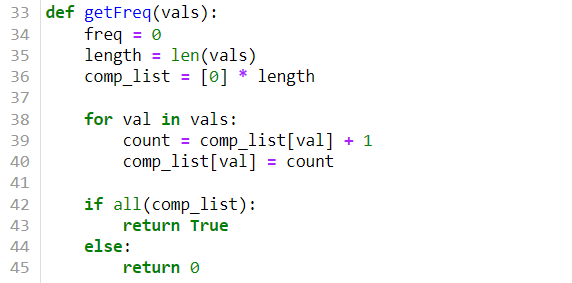
In this test, I am checking to see if a bool value of **True** is returned as the numbers in the list all appear with the same amount of frequency.



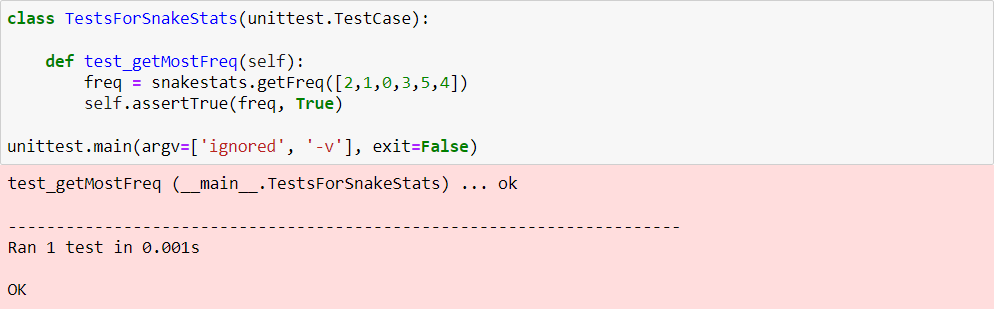
Running the test, we can see that it failed as **0 is not true**.



By changing the **getFreq()** function, a comparison list of 0s of the length of the list being tested, is created in line **36**.For each value in the test list, a count is stored in each index of the comparison list. For example, if the test list = [1,2,1,0], after running through the for loop, the comparison list will now be, [1,2,1]. However, since we are checking for the same frequency, in this test case, comparison list will return [1,1,1,1,1,1]. The if statement in line **42**, checks if all the elements in the comparison list are the same and returns True.

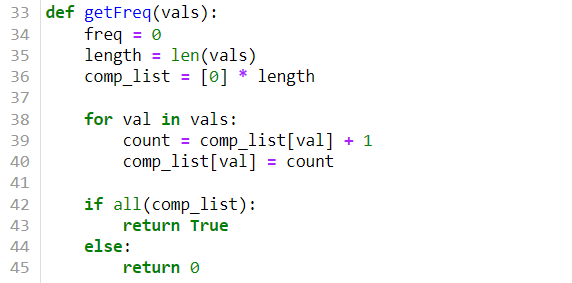


Running the test against the newest iteration of **getFreq()**, we can see that the test is now successful.

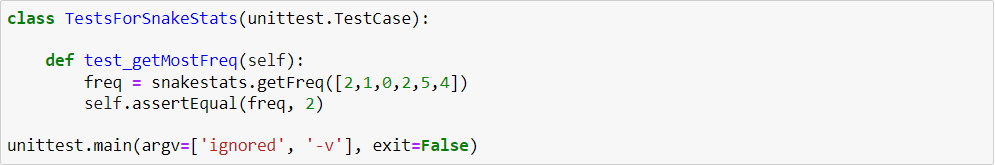


**Test Set 2: Test 2**

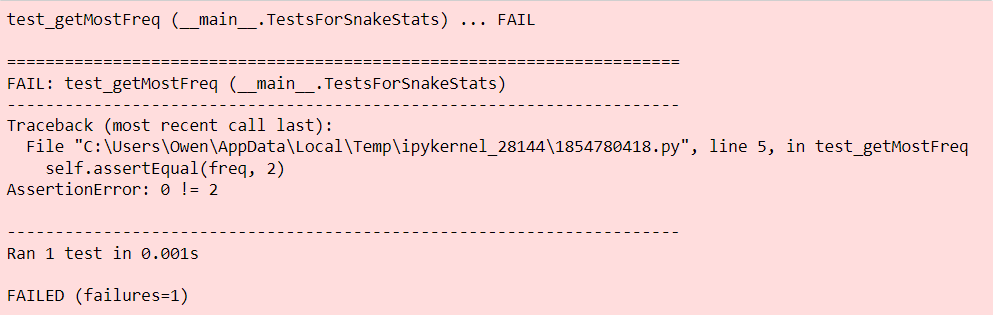
This is the current code for my **getFreq()** function.



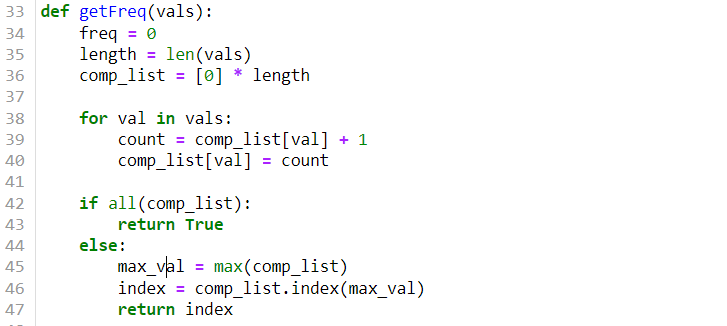
In this test, I am checking to see if the tested list contains a number with a higher frequency, will it be returned correctly. In this case, the freq variable should return 2.



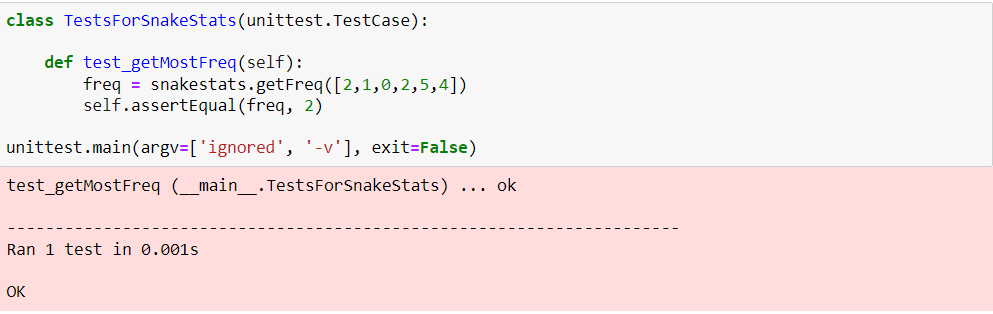
We can see that this test failed, as there was an assertion error that **0 != 2.**



By changing the **getFreq()** function if the if-condition for checking that all the elements in the comparison list are the same fails, the else-statement defaults to fetching the largest element in the list, as the index of the count stored corresponds to actual number in the test list, we can return the index with the highest count as the number that was the most frequent in the list.

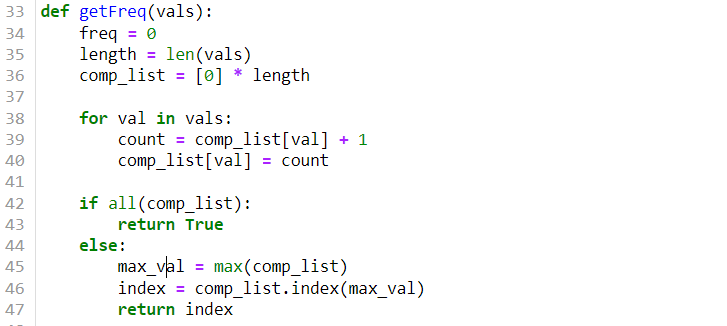


Running the same test against this new iteration of **getFreq()**, we see that the test is now successful.

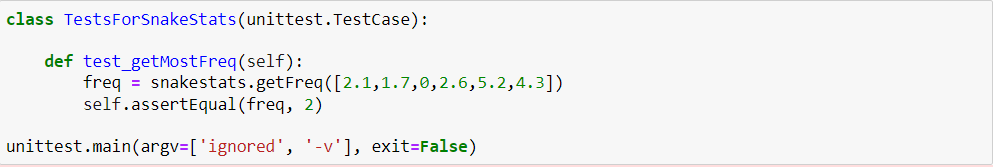


**Test Set 2: Test 3**

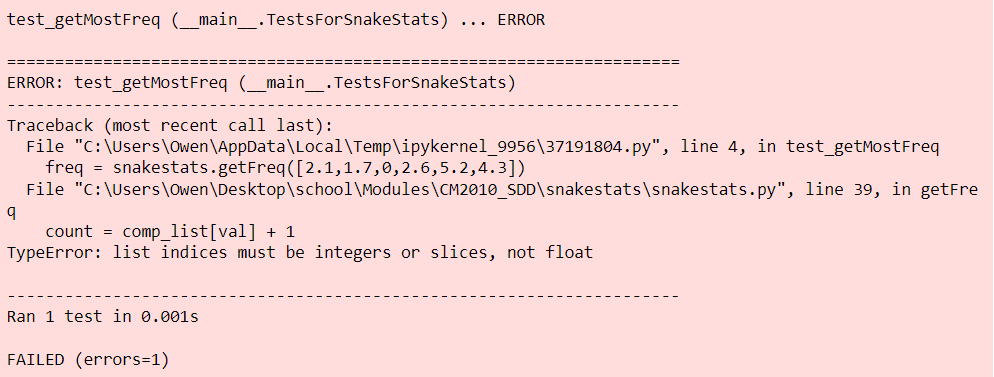
This is the current code for the **getFreq()** function.



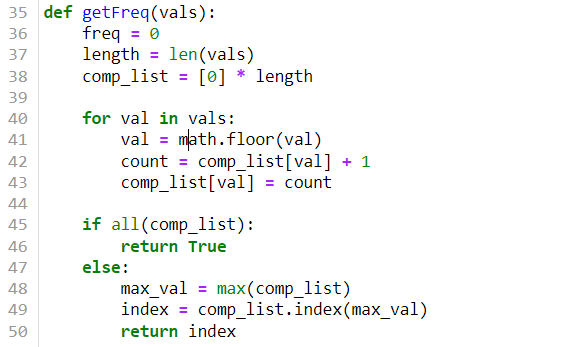
In this test, I am checking to see what happens if a list of doubles is used instead of integers.



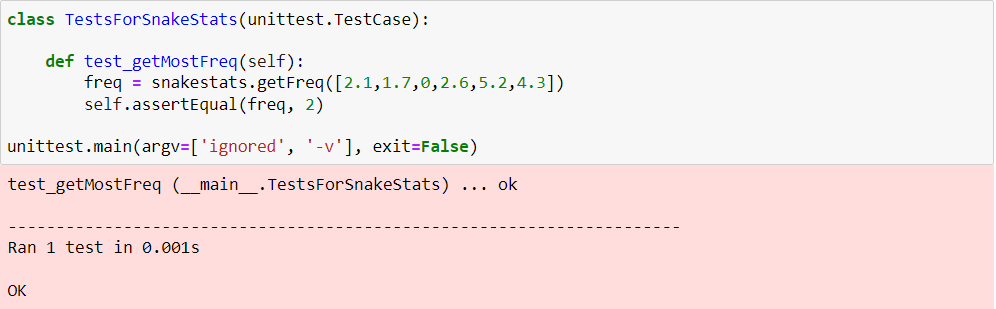
When running the test, we can see that it fails due to a Type Error. This happened as the indexes being accessed were floats and not integers.



By changing the code for the **getFreq()** function, we can try to round the floats down to their closest integer using the **floor()** function in Python. This is done in line **41**, flooring the values down before they are used as indices.



Running this new iteration of the **getFreq()** function against the test, we can see that it is now passing.



**Test Set 3: Target function name**

The third function I will be testing in snakestats is my **getMedian()** function. This function simply returns the median of a sorted list of numbers.

**Test Set 3: Explanation of strategy**

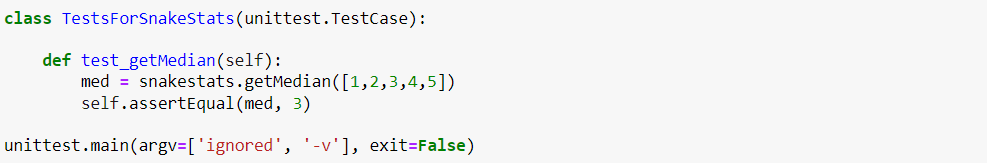
My strategy for testing is to first check if the function works when checking an even sized list, an odd sized list and an unsorted list.

**Test Set 3: Test 1**

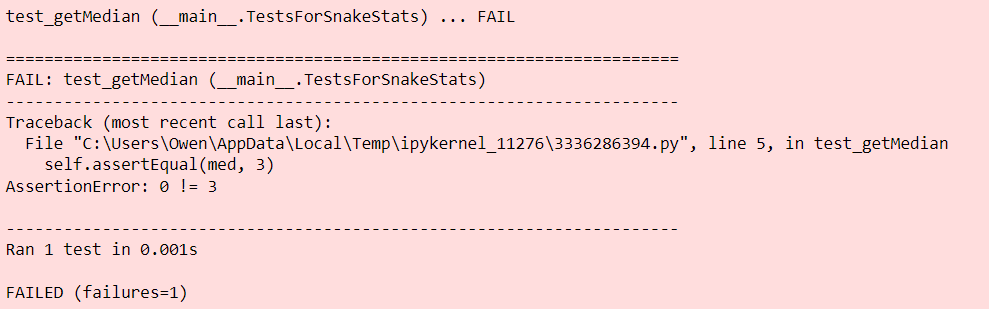
This is the current code for my **getMedian()** function.



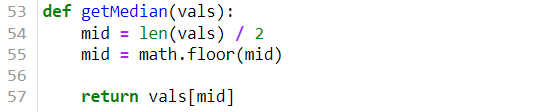
In this test, I am checking if the median 3 is returned in an odd number-sized list.



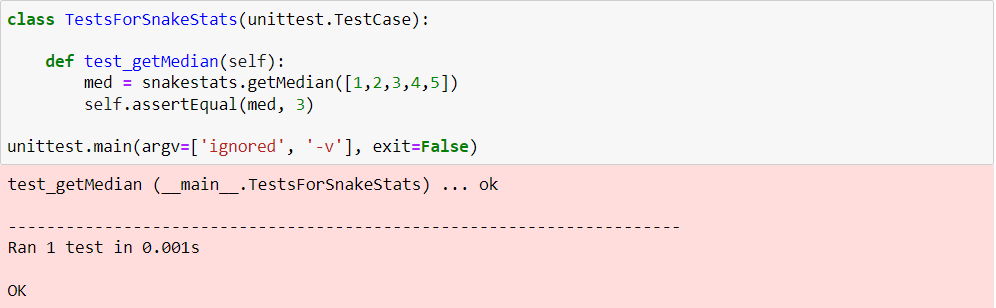
When it is run, the test fails as **getMedian()** is returning 0 and not the median which should be 3.



By improving **getMedian()**, it will now return the var that is in the position of **N (size of list) / 2.** In the case of an odd number-sized list, the value will be floored to its closest integer, as an index requires an int to work.

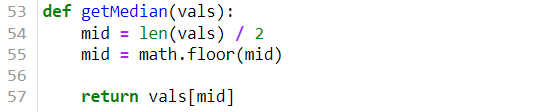


Running the test again, we see that it is now passing as the correct Boolean is being returned.

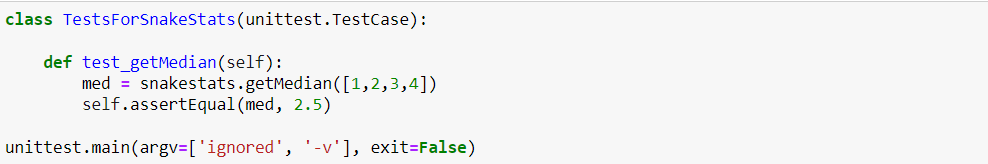


**Test Set 3: Test 2**

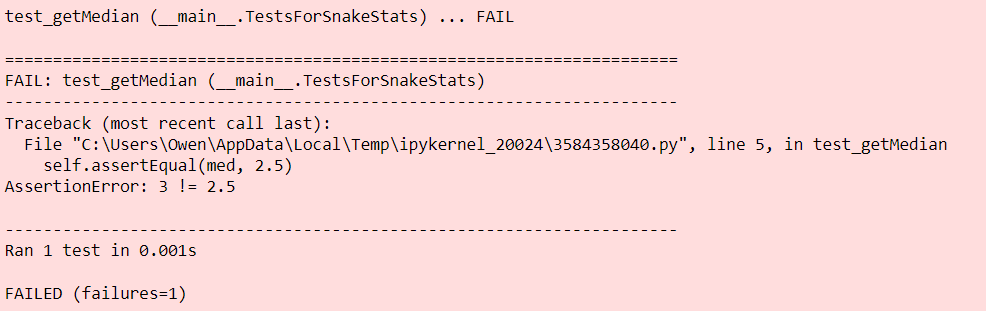
This is the current code for my **getMedian()** function.



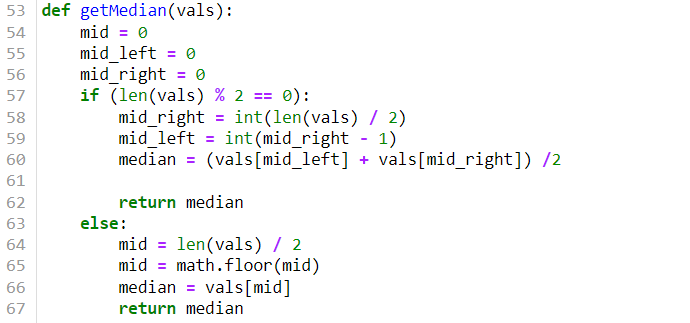
In this test, I am checking how the function responds to being tested against an even-number sized list. In the case of [1, 2, 3, 4], the median should be **(2 + 3) / 2 = 2.5**.



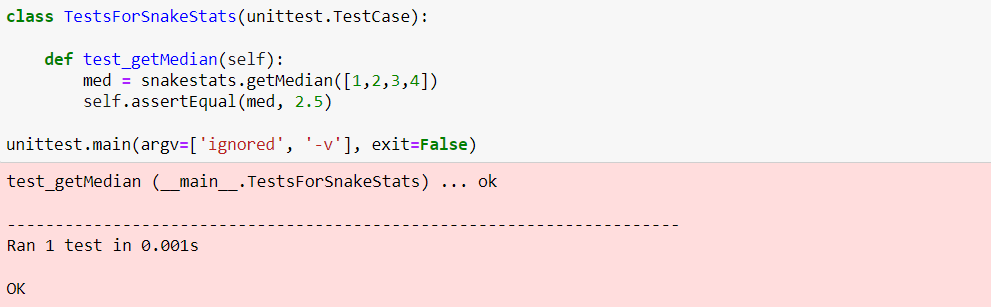
When the test is run, we can see that it fails as the median is being returned as 3.



In the case of an even-numbered sized list, we have to take the middle two elements and return the average of that. In terms of code, it would be the elements in index **N/2 and (N / 2) - 1.** By checking if the test list is odd/even, we can now take the 2 elements in the middle-left and middle-right, sum them and divide it by 2 for the actual median.

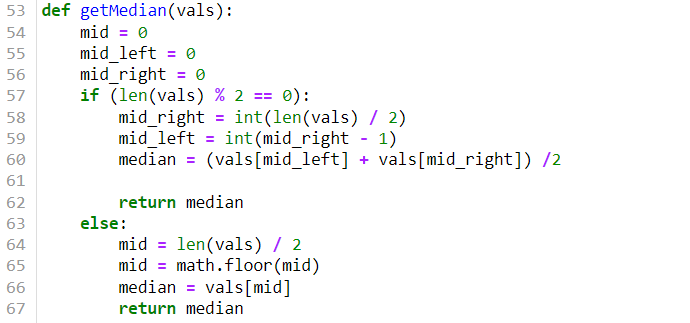


By running the test again, we can see that it is now passing.

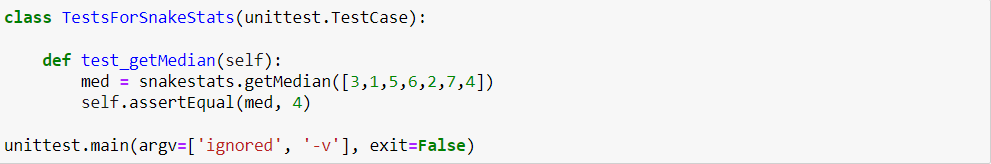


**Test Set 3: Test 3**

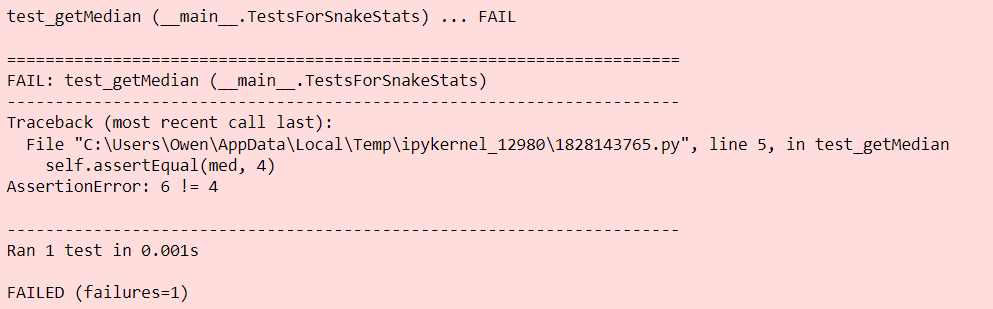
The current code for **getMedian().**



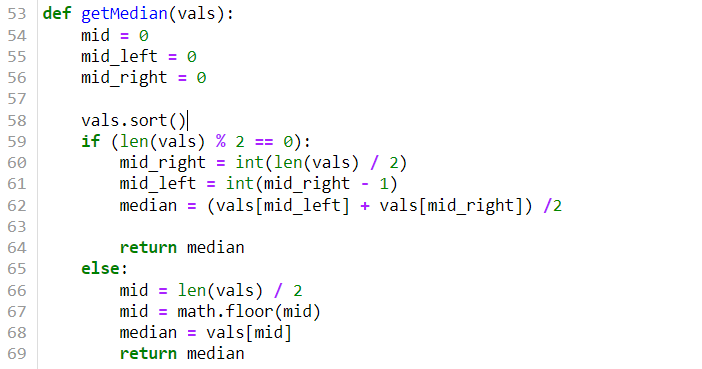
In this test, I am checking what happens if I use an unsorted list instead. as we know that median is usually calculated using an ordered data set, in this case the median should be 4.



In this test case, the test fails as it was returning 6 as the median.



By improving **getMedian()**, we can now sort the list before the median value is returned, making it so that under data set/list we use will be ordered. We can achieve this by using the **sort()** function in Python.



Running the test again, we can see that it is now passing and accepting 4 as the median of the list.

